AFRL-RI-RS-TR-2008-9
In-House Final Technical Report
January 2008

HYBRID ARCHITECTURES FOR EVOLUTIONARY COMPUTING ALGORITHMS

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-9 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

MICHAEL J. HAYDUK, Chief
Emerging Computing Technology Branch

/s/

JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* JAN 2008	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jan 03 – Sep 07

4. TITLE AND SUBTITLE

HYBRID ARCHITECTURES FOR EVOLUTIONARY COMPUTING ALGORITHMS

5a. CONTRACT NUMBER
In-House

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Daniel J. Burns

5d. PROJECT NUMBER
459T

5e. TASK NUMBER
HA

5f. WORK UNIT NUMBER
EC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFRL/RITC
525 Brooks Rd
Rome NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITC
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-9

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-0064

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report documents the results of an in-house project aimed at identifying, developing and evaluating applications of evolutionary computing methods to hard optimization problem test cases on a single PC computer, a cluster of computers, and hardware FPGA platforms. We surveyed the evolutionary computing literature and chose to focus on the Genetic Algorithm (GA). We applied the GA to Non-Linear Coupled Ordinary Differential Equation (ODE) Parameterization, the DNA Code Word Library Problem, and the Networked Sensor Power Management Policy Problem. The first problem used an ODE biomodel for Antigen-Antibody binding, and we demonstrated speed-ups on the order of 100-1000x by moving from interpreted languages to compiled C. We parallelized this C code using the Message Passing Interface (MPI), and demonstrated linear speed-ups on a cluster. A GA solution for the DNA Code Word Library Problem was also parallelized, and was faster than any algorithm found in the literature. We also developed hardware accelerated prototypes for the GA for this problem that achieved speed-ups on the order of 1000x. These prototypes used random and rank based selection, single point crossover mating, a declone operator, systolic arrays for the LLCS and Gibbs energy metrics, a multi-deme GA, and exhaustive search for producing locally optimum codes.

15. SUBJECT TERMS
Genetic Algorithm, Optimization, Ordinary Differential Equation, parallel, distributed, Field Programmable Gate Array, hardware acceleration, DNA, Codes, Gibbs energy

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Daniel J. Burns
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	150	**19b. TELEPHONE NUMBER (*Include area code)*** N/A

TABLE OF CONTENTS

List of Figures

List of Tables

1.0 Introduction

This final technical report documents the results of work performed under an in-house project that investigated novel computing architectures that facilitate the application of stochastic evolutionary computing (EC) algorithms to hard, NP-complete optimization problems. This project was undertaken at a time when an increasing number of researchers were successfully applying such methods in a number of diverse problem domains.

While we focus here mainly on the results of the last phase of the project, spiral 3 (hybrid software/hardware acceleration), for completeness, in this section we briefly summarize the project goals, main tasks, and results of spiral 1 (workstation software), spiral 2 (distributed software on workstation cluster), and the early work during spiral 3 (hybrid software/hardware acceleration). Further detail on the earlier work may be found in a previously written in-house interim technical report that is available through the Defense Technical Information Center [1]. Then in Section 2 we describe the main work of spiral 3, i.e. progress on building and testing single chip prototypes that were hardware accelerated versions of a particular genetic algorithm (GA) along with an optimization problem application. Section 3 discusses the results of testing done with various prototype versions, and Section 4 concludes and discusses some suggestions for spin-offs and future work. Section 5 acknowledges those who contributed to this work, and Reference and Appendices of papers published and presented during this project follow. Throughout this technical report, we have made an effort to write about the work and its significance to inform readers with a variety of technical familiarity.

Spiral 3 investigated the hypothesis that simple EC methods like the genetic algorithm (GA) might offer advantages in terms of achieving extreme speed-ups by implementation in hardware. This typically would not be obvious or appreciated by problem domain experts working only on software solutions. We know that hardware implementations are often highly parallelized, and many EC algorithms are ‘embarrassingly parallel’ to begin with, meaning that many sets of repetitious calculations involving the fitness function may have to be performed over and over again for a large number of individuals in a population. If the calculation in the problem fitness function (and GA operators) involve simple mathematics (e.g. Boolean and integer operations), and if they can be pipelined in a fast data path, hardware acceleration may yield significant speed-ups (i.e. >50X). The results of spiral 3 have clearly shown this to be the case for a very difficult DNA Code generation problem that is of current interest to us, by demonstrating extreme speed-ups of up to 1,000X in prototype single chip Field Programmable Gate Array (FPGA) designs that solve this problem for the first time in an all hardware approach.

This project was carried out in the Advanced Computing Division of the Information Directorate (RI) of the Air Force Research Laboratory (AFRL) at the Rome Research Site, Rome, NY, with limited support of the principal investigator’s time by Defense Advance Research Project Agency (DARPA) project management funds, and also with contributions by summer faculty and students.

1.1 Project Goals

One goal of this project was to determine whether EC algorithms offer any advantages over more classical methods, especially in the context of parallel and hybrid (or heterogeneous) hardware/software implementations that are aimed at achieving extreme solution time speed-ups and problem size scaling. In general, this project investigated ideas aimed at making progress in the area of improved solution engines for NP-hard optimization problems that are relevant to a number of RI mission area applications.

Another goal was to synergize with other ongoing in-house and summer faculty research topics, as well as Air Force Office of Scientific Research (AFOSR) and DARPA programs managed by RI, and to identify target problems in ongoing programs that might benefit from the development of new optimization tools for solving hard computational problems. We accomplished this goal by working on problems drawn from these programs, in particular, work during the third spiral that focused on a difficult test case problem called the DNA Code Library Generation problem. This problem is directly related to technical areas of interest to RI that relate to computing with bio-molecules, and nano-scaffold self-assembly for bio- and nano-electronics, and it has been the subject of recent AFOSR sponsored efforts proposed and managed by RI. We also did involve at least 3 summer faculty and 2 summer students during the course of this project.

Finally, throughout the entire project, two other goals were to raise the level of awareness of workers at RI about present day research and applications of evolutionary computing methods, and to mine the RI mission areas for additional candidate problems that could potentially benefit from this knowledge and the results of this effort.

1.2 Summary of the Accomplishments of Spiral 1 (software on workstation)

During Spiral 1 we surveyed the evolutionary computing literature, attended several relevant conferences (e.g. the Genetic and Evolutionary Computing Conference), and we sponsored a few on-site lectures by representative experts in EC. These activities helped us to identify the genetic algorithm as perhaps the most general purpose and widely used EC algorithm, and we decided to use the GA as a test case algorithm to carry through all 3 spirals of this effort. During spiral 1 we developed a number of software prototype optimization tools that used the GA, in 3 popular programming environments (Labview, MatLab, and compiled C), and we evaluated their performance relative to alternative approaches when they were applied to a limited number of test case problems that were of interest to us. The first test case problem was parameterizing a particular bio-model that consisted of a set of Non-Linear, coupled Ordinary Differential Equations (ODEs). This problem was of wide interest to workers in certain biologically oriented DARPA programs AFRL Advanced Computer Architectures Branch (RITC) managed at the time. The bio-model described antigen-antibody binding at surfaces, and was supplied by a principal investigator (PI) in the DARPA Simulation of Biological Systems (SIMBIOSYS) program [2]. The basic problem involved fitting model parameters so the model would properly predict experimental data. Previous approaches by the PI involved MatLab optimization methods that did not converge, and it was necessary to reformulate a simplified, linear version of the model, and to use a step-by-step estimation procedure to obtain workable estimates for the several parameters. This was slow, and we found that a simple GA could quite easily fit multiple parameters, even when working directly with unreduced non-linear models, and for sparse and noisy experimental data. The following list summarizes the activity of Spiral 1.

Spiral 1: (PC platform - complete)

- Translated Labview version GA optimization tool and Ag/Ab binding bio-model to C
- Obtained MatLab based Genetic Algorithm Optimization Toolbox from North Carolina State University, integrated MatLab bio-models from Purdue University SIMBIOSYS PI and also with C version bio-model derived from Labview version.
- Ported Purdue MatLab bio-model to C, integrated with C version.
- Evaluated the speed, accuracy, convergence, and scaling performances of the Labview, MatLab, and C versions of GA ODE Parameterization tool
- Evaluated Virginia Tech GEPASI bio-model simulation and fitting tool
- Developed Java Open Agent Architecture (OAA) wrapped compiled C version for contribution as BioSpice agent under the DARPA BIOCOMP program
- Developed web browser interface version for use by remote, non-programmer users
- Established Evolutionary Computing Interest Group at RI and hosted 7 speakers.

Figure 1 shows an example of the speed-up results obtained by simply porting the GA bio-model fitter from the initial Labview and MatLab versions to compiled C versions. Note that the y axis of the chart shows the time required to run 100 generations of the GA fitter, for various GA population sizes. The final, fastest compiled C versions (lowest curves) used a hand-coded GA, bio-model, and ODE solver (rather than packaged GA and ODE libraries), and they achieved speed-ups of 100x - 1000x over the earlier versions (upper curves). The PI's original non-automated, reduced order model fitting method was not scripted, and was so slow that it would not appear in the chart of Figure 1. An important point to make here is that good solutions to this problem took hours using previous non-GA methods, about 5 hours using the early GA versions, but only about 20 seconds in the final versions.

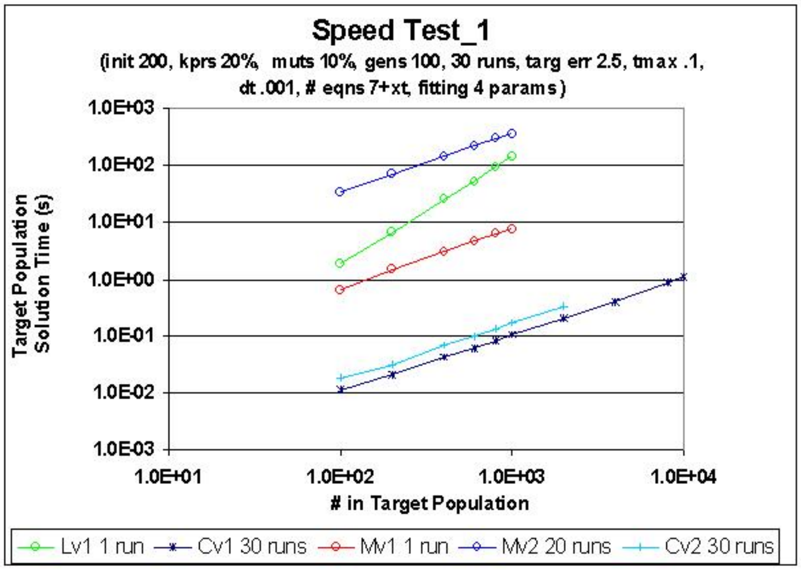


Figure 1. Speed Test_1, solution time of the target population.

We suspect that in general there is a tendency on the part of problem domain experts to disregard non-familiar approaches if they are used to working with particular tools in a particular computing environment. This result demonstrated that game changing performance can be had really quite easily, by simply hand-coding the problem, and using a more capable optimization algorithm. Throughout this project we pressed this point to this PI and others in the SIMBIOSYS program, both during PI meetings, during site visits, and even more widely to PIs in the later DARPA BIOCOMP program. Eventually we did in fact see the BIOCOMP program develop sophisticated, work-flow based bio-model parameterization and bifurcation fitting tools that used both GA and parallel direct search methods to fit large non-linear models [3]. Also, our in-house evaluation of fitting methods available in the GEPASI bio-model tool [4] showed for our bio-model, GEPASI's "evolutionary" and "random" fitting methods were the only ones that did converge [1, p. 19]. The other optimization methods (Hook and Jeeves, Levenberg-Marquardt, Levenberg-Marquardt multistart, Nelder and Mead simplex, and Simulated Annealing) did not converge. Further, the GA method was very much quicker than the "random" method, as would be expected. This point provides relatively independent confirmation of our own conclusion that a GA based fitter can be effective and efficient for parameterizing full, non-linear bio-models.

It was also interesting to do a quick literature search on this problem at the end of this project. A Google search on "direct search genetic algorithm", "genetic algorithm ODE" and the like produced a number of references, in addition to our own. For example, a good discussion of the importance and difficulty of the nonlinear ODE fitting problem can be found in [5,6,7], which also describe and illustrate the use of iterative, direct search methods for non-linear models. Two possible objections to these methods that may limit scaling and speed are that some require the calculation of gradients (which are computationally expensive), and while some do not, they may require large amounts of memory to keep track of promising remaining areas in the search space that need to be explored (which may grow exponentially). Finally, at least two references [8,9] also specifically compare the performance of the GA with direct search and classical methods, and both conclude that a well tuned GA is as good or better than well tuned other methods. To some degree this would be expected, since the 'No Free Lunch' theorem of Wolpert and Macready [10] holds that on a particular problem, different search algorithms may obtain different results, but over all problems, they are indistinguishable. That is not to say, however, that for certain problems, or for certain tunings of its search parameters, one algorithm cannot outperform another algorithm. Indeed, that often seems to be the claim of many studies involving evolutionary methods. Even so, closer inspection often reveals that it may only be necessary to change the algorithm parameters or the problem to reverse the conclusion (or to have the GA perform even better)! In the end, there are some characteristics that may doom an algorithm or a problem's prospects for speed-up by parallelization or hardware implementation, including complex and floating point mathematical content, large local memory requirements, and anything that causes large communication bandwidths between nodes or processes.

In summary, in spiral 1 we found that a GA based optimizer could solve the Non-Linear ODE Parameterization problem for our test case problem, and that it was superior to common methods currently being used by some DARPA program PI's because it worked on full, unreduced, non-linear models, and can easily deal with sparse and noisy data of the sort provided by real world experiments, and in our case it operated much faster as well. We also developed two versions of the tool with enhanced user interfaces. One is a BioSpice agent version, and one is a web browser interface version. The work in this spiral was presented to the 2003 Scientific Advisory Board during their visit to RI, in the Advanced Computer Architecture focus area poster session. It was also discussed with relevant DARPA SIMBIOSYS and BIOCOMP program PI's at various PI meetings, and it was reported at an evolutionary computing conference [11].

1.3 Summary of the Accomplishments of Spiral 2 (distributed software on workstation cluster)

During the second spiral we started by basically parallelizing the C code version of the GA based bio-model parameterization tool, by adding Message Passing Interface (MPI) communication to implement a distributed GA that ran on a cluster of workstation computers. We evaluated both a Farming Model GA, in which there is one population, and fitness function evaluations are farmed out to a number of processors, and an Island Model GA, in which separate populations are evolved on a number of processors, with periodical migration of a few good individuals around a ring of processors. The Island Model GA worked very well, and achieved approximately the expected linear speed-up. We also applied it to two additional optimization problems of interest to workers in RI, the Networked Sensor Power Management Policy Problem, and the DNA Code Word Library Generation Problem. In these studies, we also evaluated our tool's performance relative to the best non-GA methods found in the literature. The following list summarizes the activity of Spiral 2.

Spiral 2: (Cluster platform - complete)

- Developed Distributed Farming and Island Model GA applications to Non-Linear ODE parameterization (C/MPI), evaluated performance scaling vs. # processor nodes.
- Developed 2nd application of Distributed GA to DNA Code Word Library Generation problem and demonstrated linear speed-up performance scaling vs. # processors nodes.
- Developed 3rd application of GA to Networked Sensor Power Management Problem
- Visited the Air Force Institute of Technology (AFIT), Wright State University, Virginia Tech. to discuss collaborations.
- Attended the Genetic and Evolutionary Computing Conference (GECCO) and Military Applications of Programmable Logic Devices (MAPLD) Conferences and presented spiral 2 results.

The main results of this spiral for the ODE Parameterizer problem are shown in Figure 2. The ideal and measured speed-up curves are shown for solving the problem using different numbers of processors in the cluster. Again, the Island model GA was designed to evolve separate populations at each processing node, and to pass a few good individuals around the ring of processors after epochs of a few generations. While the speed-up curve for the Island Model GA (middle curve in Figure 2) is not perfect (top Ideal Linear Speed-up curve), it is much better than the Farming Model GA (lower curve), and there is no drastic slow down plateau apparent up to about 29 processors.

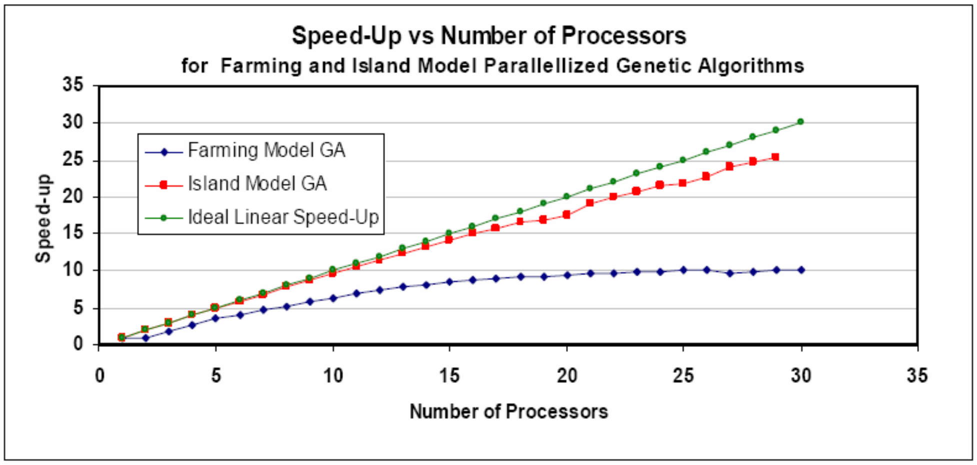


Figure 2. Speed-up curves for distributed GA ODE Parameterizer versions.

Surprisingly, on-line searches for "ODE parameterization distributed GA" and the like turn up only our own work. Farming model distributed GA's are available in the literature, but not a lot of work on distributed GA's similar to our own. This leads us to believe that it may be rather unique, and we are fairly certain that it is a novel and unique approach for solving the DNA code problem, which we describe next.

During this spiral we developed both a single PC version and a distributed Island Model version of a second application of a GA optimizer to the *DNA Code Word Library Generation Problem*, which is of current interest to workers in RITC and elsewhere. This problem involves composing highly constrained sets of Watson-Crick pairs of short DNA oligo-nucleotide strands, e.g. about 16 base pairs long. The pairs in the set each consist of two strands that are perfect complements that bind, or cross-hybridize well to each other, but poorly to their own reverse complements (RCs) and any strand in any of the other pairs in the library. This is a hard problem that is known to be NP-complete, and at least 4 University groups are actively working on it because there are important applications of such DNA Codes to the design of bio-assay micro-array chips, self-assembly of nano-structures, and schemes for data storage and computation using bio-molecules. Random search and exhaustive search have proven ineffective for building large libraries, and current techniques use stochastic and heuristic methods.

Our approach to this problem starts building a library by finding one pair using random search. It then breeds additional words using a GA guided by a multi-objective fitness function that measures the string edit distance (calculated by the Levenshtein Martix) and that also counts the number of pairs presently in the library that reject a given candidate pair. The GA uses an efficient mutation heuristic that chooses a base pair to mutate at random, checks the fitness of words with all possible single base changes at that position, and uses the mutation that improves fitness the most. The populations may also be decloned periodically to help widen the search.

Figure 3 shows the results of using a distributed GA solver for this problem, as well as comparison to results obtained for two non-GA algorithms found in the literature (Markov and Stochastic). The curves show the average time required to discover words, for multiple runs of the algorithm (lower is better). The upper curve (Stochastic) is similar to our GA, but it starts with a random library of DNA words that do not satisfy the required non-cross hybridizing constraints, and tries to improve them by mutations. Our GA starts with an empty library, breeds better word candidates, and adds good words to the library only as they are found. Thus, the Stochastic method must do many more constraint evaluations from the beginning, and is slow compared to GA

The lower curves in Figure 3 compare the performances of Markov and GA, for the case of 1 and 16 processors used in the cluster. In both cases GA actually finds words faster up to the time at which words become very difficult to find, at which point both algorithms turn up and have difficulty finding the last few words. We also note that algorithm parameter tuning does effect these results, and that to date Markov has actually found slightly larger libraries than GA, but GA consistently finds a large portion of the total number of words that can be found faster.

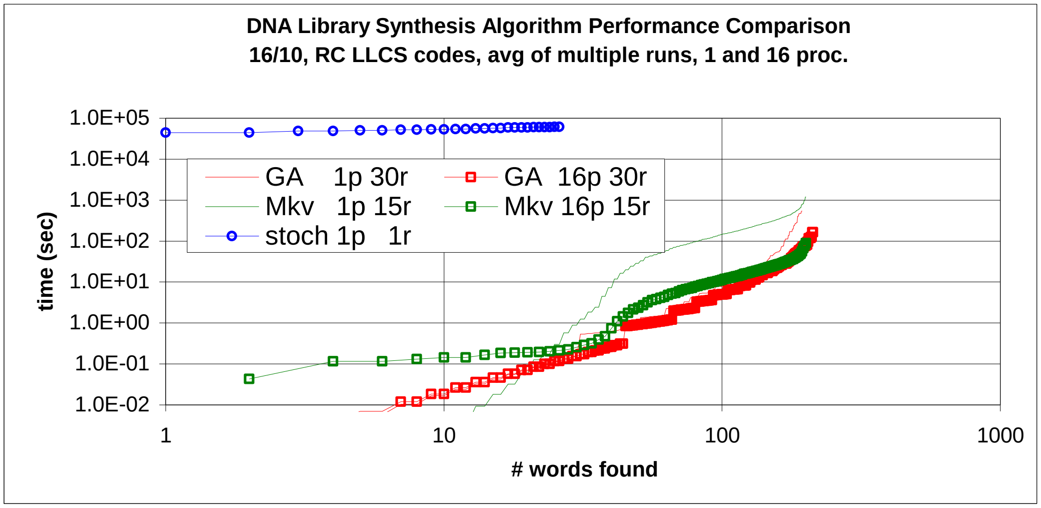


Figure 3. Comparison of Markov, GA, and stochastic DNA Code Word Library Generation methods.

In summary, during spiral 2, a distributed Island Model GA optimization solver was developed and successfully applied to three problems: Non-Linear ODE Parameterization; DNA Code Word Library Generation; and Sensor Network Energy Management. This distributed GA version exhibited good speed-up scaling vs. number of processors on a 30 node cluster. The GA based DNA Code Word solver exhibited performance that we believe rivals the best known algorithms in the world for this problem. The work done on the GA/DNA Codes application during this spiral was reported at a conference on nano-self assembly [12], and at a conference on evolutionary algorithms [13-15]. The work on evolutionary optimization of sensor network power management was reported at a conference on evolutionary computing [16], and at a conference on low power electronics [17].

1.4 Summary of the Early Accomplishments of Spiral 3 (hybrid software/hardware acceleration)

During the first part of spiral 3 we began developing a hardware accelerated version of a GA optimizer and the DNA Code Word Problem. We chose to go forward into this spiral with the DNA Code Word Library Generation problem because it is an integer problem. First we spent some time investigating Higher Order Language (HOL) tools for translating C to Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), our preferred language for designing hardware logic. We worked with one supplier's tool (Impulse C) almost to the point of success, i.e. we were able to design and simulate an array calculator for the Length of the Longest Common Subsequence (LLCS) that implemented the heart of the fitness function calculation. However, we were not able to produce a VHDL version of the entire C application that included the GA, DNA Library Generator, and LLCS calculator automatically from the tool.

We then hand crafted a ripple through version of the LLCS systolic array that synthesized with an expected clock frequency of 10MHz, which represents a speed-up of 100x over the software C version. We also designed a 2D pipelined systolic array version that synthesized with an expected 80MHz clock frequency (close to the targeted 1000x speed-up). We also determined that the LLCS calculator array took less than 20% of the target FPGA chip resources, which suggested that there were excellent prospects for fitting the entire GA/DNA Code application into one FPGA chip. The following list summarizes the work in the early part of spiral 3.

Early part of Spiral 3 (hybrid software/hardware acceleration)

- Identified, purchased and installed VHDL software development tools, reconfigurable logic hardware platforms, and FPGA synthesis tools
- Collaborated with Impulse, Inc. evaluating Co_Developer C to VHDL translator tool
- Preliminary design completed for GA optimization algorithm FPGA core written in VHDL for hardware implementation aimed at extreme speed-up.
- Final design in VHDL of a Levenshtein Matrix systolic array calculator for hardware acceleration of the fitness function evaluation for the DNA Code Generation Problem.

An example of the main results of the early part of spiral 3 are shown in Figure 4 and Table I. Figure 4 shows a high level functional block diagram of the first version of an overall application prototype. The design included a test bench to simulate the host PC while initializing and controlling the application, a number of on-chip SelectBlock memories (BRAMs) to hold the GA population, the fitness values, and the DNA Library words. It also shows an entity called MemBlock.vhd for sequencing the fitness evaluation of each population individual against all words in the library, and for storing the results. Finally, it contains an entity called Spit_Me for streaming a set of operands into the 2D LLCS systolic array pipeline calculator called Do_CheckerMatrix. The design of Figure 4 was simulated and shown to operate correctly, and was synthesized successfully.

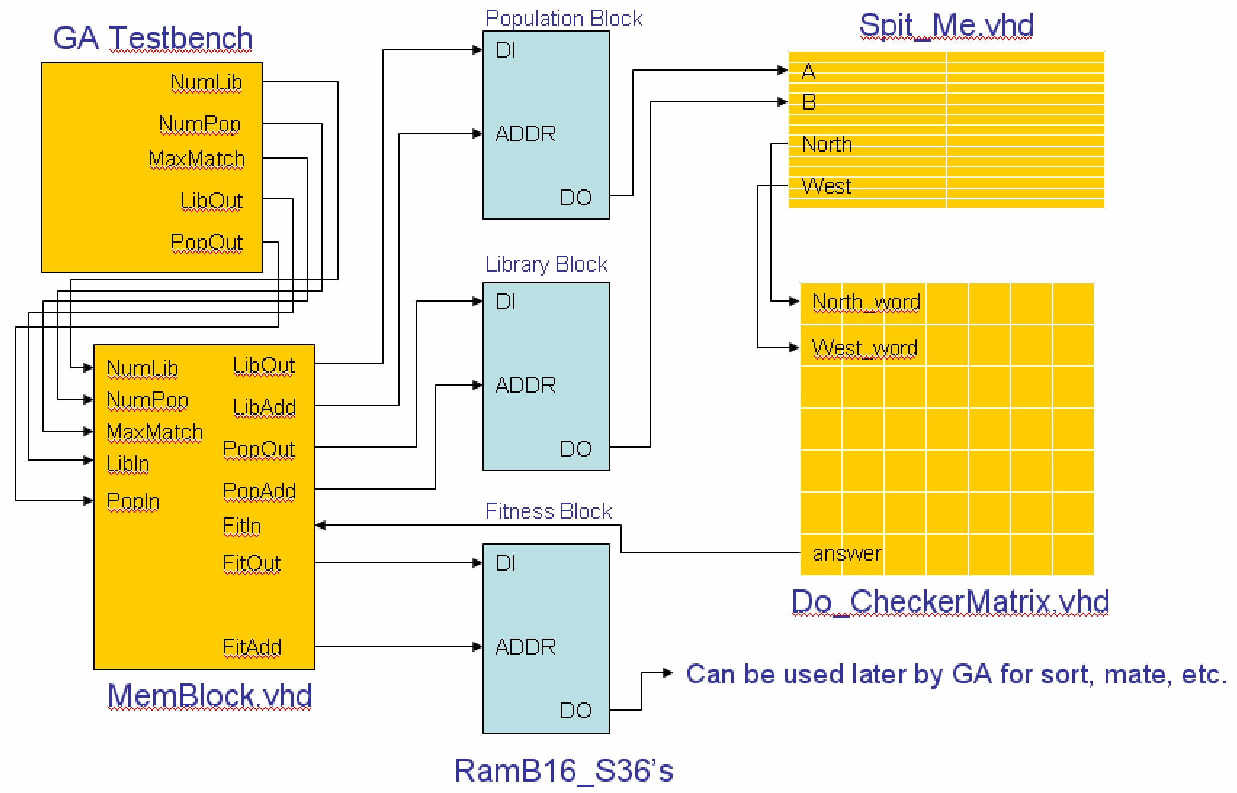


Figure 4. Upper level functional block diagram of fitness evaluator.

Table I shows the synthesis run report for the Do_CheckerMatrix.vhdl LLCS systolic array, for the case of 512 population size and 16 mer (32 bit) DNA word libraries. It is clear that much larger populations and libraries could be used, since only 3 of the 144 SelectBlock RAMS were used for this case. Since our target FPGA platform had on the order of 35,000 Logic Units (LUTs), it appeared that systolic array would use about 20% of its resources, with about 80% available for the GA and DNA Code Word Library application, which was very encouraging. This also indicated that it might even be possible to support multiple fitness evaluators and even multiple GA populations on one FPGA.

Table I. Synthesis resource utilization report for Do_CheckerMatrix.vhdl

Number of Slices:	4283
Number of Slice Flip Flops:	2544
Number of 4 input LUTs:	7532
Number of bonded IOBs:	98
Number of BRAMs:	3
Minimum clock period:	12.37ns (80.8MHz)

At this point, much of the GA and DNA Code building parts of the main program were still blocks of sequential behavioral VHDL, rather than parallel clocked processes that would synthesize and operate in an efficient manner. In fact, although they simulated correctly, they did not synthesize, which highlights a shortcoming of today's synthesis tools. (We think they should be able to synthesize such blocks of correct behavioral code). The next steps were to re-write these parts of the application as clocked parallel processes, and to integrate the entire GA core, DNA Code application, and LLCS fitness function evaluator into one FPGA and test it. In the next section we review progress on those tasks and follow-on work that demonstrated several versions with various improvements and experimental features.

Finally, papers describing the early work in spiral 3 were give at an AFRL conference on algorithms [18], and a conference on reconfigurable logic applications [19].

2.0 Technical Approach - spiral 3 (hybrid software/hardware accelerated platform)

In this section we briefly comment on the suitability of the DNA problem for use as a test case problem in this spiral, and we mention some architectural alternatives and their limitations imposed by the platforms we had available for this work. We also briefly discuss the software design tools and hardware platforms used during this project. Then we describe the main design features of each of a series of single chip FPGA prototypes that we actually implemented and tested. These are all the first known examples of hardware accelerated versions of the LLCS fitness metric in a 2D systolic array, and the overall GA/DNA Code application problem. The main accomplishments of spiral 3 are given in the following list, and details are provided in the sub-sections that follow. Test results obtained with these prototypes are given in Section 3.

Spiral 3 (hybrid hardware/software accelerated platforms)

- Designed hardware FPGA cores for operators of general purpose and multi-deme GAs
- Designed the first known hardware DNA Code Library generation application
- Designed the first known hardware systolic array calculators for DNA code word hybridization metrics, including the Length of the Longest Common Sub-string (LLCS) and the stacked pair nearest neighbor model Gibbs Free Energy of Binding
- Designed and tested a series of 4 prototype FPGAs that integrated various combinations of hardware genetic algorithm and metric calculators with the hardware DNA Code Library building application, as well as exhaustive search versions for extending codes
- Published several papers and made several presentations at Conferences that spanned a wide range of topics, including reconfigurable logic, evolutionary computing, DNA computing, computational intelligence for bioinformatics, and bio-threat detection.

2.1 General problem and platform considerations for hardware acceleration

The DNA Code problem is a particularly good candidate for acceleration in hardware because it involves simple 32 bit integer and Boolean math throughout. While this is true of a number of other problem types as well, it is not generally true of all problem types. So, the hardware GA and its operators are most applicable to other problems that are compatible with hardware acceleration, with appropriate modifications to the genotype (the variables representing solutions to a problem), and fitness function (the model of the problem and constraints that are evaluated to measure how well each individual in the population solves the problem). Floating point calculations are generally thought of as a factor that limits prospects for hardware acceleration. Although both single and floating point FPGA cores are available, they do take up resources, and cost time if conversions must be done from integer to floating point and back in a data path. However, many floating point problems can be approximated well enough using fixed point methods that use integers to represent and calculate on discrete floating point values over a limited range.

Speeding up the DNA Code problem would be very valuable because solving it actually requires solving a series of NP-complete problems, each with increasingly difficult constraints on cross hybridization that accumulate as words enter the library. We started this speed-up effort by first profiling the execution time of the overall application and its subroutines, and we found that over 98% of the solution time was spent evaluating the LLCS cross hybridization metric in the fitness function. At first we targeted that calculation for efficient hardware implementation, but it became obvious that even if we could reduce the LLCS calculation time to nearly 0, the execution time of the speed of the overall application would then be dominated by the GA control loop and its operators. Clearly, we would have to implement the GA and its operators in hardware as well. Various other hybrid implementations are possible, e.g. GA in software on the host (or an embedded) PC with LLCS metric in hardware, and these might actually have been easier to design and implement. However, extreme acceleration can best be achieved by a 100% hardware solution. The main reason for this is that communication between the host PC and the FPGA hardware processing element (PE) takes place across the host system internal data bus. The workstations used for this project used the PCI protocol (32 bits at 33MHz), the PCI-X protocol (64 bits at up to 133MHz), or the PCMCIA/PCI protocol (32 bits at 48 MHz) to communicate with the FPGA cards. Also, other uses of the system bus by the host operating system may introduce delays and bus latencies that are significant. A 100% hardware solution minimizes Host to PE communication and results in better speed.

Finally, at this writing, multi-core processors and the Cell Broadband Engine^{TM} have only just recently appeared, and they were not evaluated as execution platforms during this project. It would appear that both would be good candidates for implementing highly parallelizable EC algorithms, although they both lack a reconfigurable logic capability that might better support the 2D systolic array used in our approach.

2.2 Software tools for hardware development

In general, the process of rewriting the C application in VHDL started with the C code version from spiral 2. We stripped out the MPI calls that implemented a distributed GA, and re-coded it in behavioral VHDL, i.e. in blocks of sequential, or behavioral VHDL code that closely mirrored the program flow of the C version. A number of separate parallel processes were then pulled out of the main program code to implement portions of the application in clocked VHDL code (e.g. mutation, mating, decloning, etc). Following this, the main control program was recoded as a clocked VHDL process. Finally, a C host program was written to interface with the user to allow specification of problem and GA variables, initialize the PE, monitoring progress during runs, and receiving a final reporting of results at the end of a run or series of runs. In general, there was significant communication between the host and PE at the beginning and end of runs, but very little communication during runs.

We used the Mentor Graphics ModelSim XE III tool [20] for code development and simulation. This tool enables one to observe the operation of the code in terms of signals and data values that flow through the control and data paths. This can be done in two ways, by inserting statements that cause messages and results to print in a monitor window during simulation, and by setting up and observing various sets of signal waveforms vs. time as the simulation executes.

At various times parts of the code, and later the entire application, were synthesized using either the Synplicity Synplify tool [21]. We also used the Xilinx Integrated Software

Environment (ISE) Version 8.1 set of tools [22] to synthesize parts of the application and check the expected execution speed and the required amount of resources.

In general, we used two versions of the tools during this project: free versions of all three tools that were hosted on a desktop workstation (that are available for download on the web for evaluation); and an expensive fuller versions that were hosted on one of our internal servers (Gonzo). Summer faculty workers who contributed to this project also used academic versions of all three of these tools, both at AFRL and at their University. The free version of ModelSim was crippled, which means that it executed very slowly when the number of lines of code exceeded a certain limit (e.g. 5,000 lines). This made it impossible to simulate multiple, long runs of the entire GA/DNA Code application, e.g. to determine the average number of words that can be found by a run. It was impractical to simulate runs longer than a few hundred milliseconds of simulated FPGA execution time, because that took many minutes. This meant that the only practical way to test the application was after synthesis in hardware. For example, we later typically used hardware GA run times of about 5-10 minutes, and hardware exhaustive searches that took about 2 hours, which would be impossible to simulate in ModelSim. We did not use a hardware in the loop debugger in this work. This was somewhat troublesome because we did experience bugs at times that simulated OK in ModelSim, but did not operate properly in when synthesized and tested on the FPGA.

Another useful design tool that we used both on paper and in the Xilinx ISE tool set was a simple graphical state diagram editor (StateCAD). Experienced VHDL designers may just sit down and write code, but for complex designs with many parallel interacting processes it is often helpful to make state diagrams of each process. These diagrams resemble flow charts that C programmers may be familiar with, but instead of sequences of calculations, they specify the inputs, state machine transitions, and outputs of each process in a graphical manner. The designer can then manually code VHDL from the diagram, or have the tool automatically generate VHDL in some cases. For illustration purposes, Figures 5-7 show examples of a state diagram, VHDL code derived from it, and a waveform debugging display, respectively, for the declone operator coded for this project.

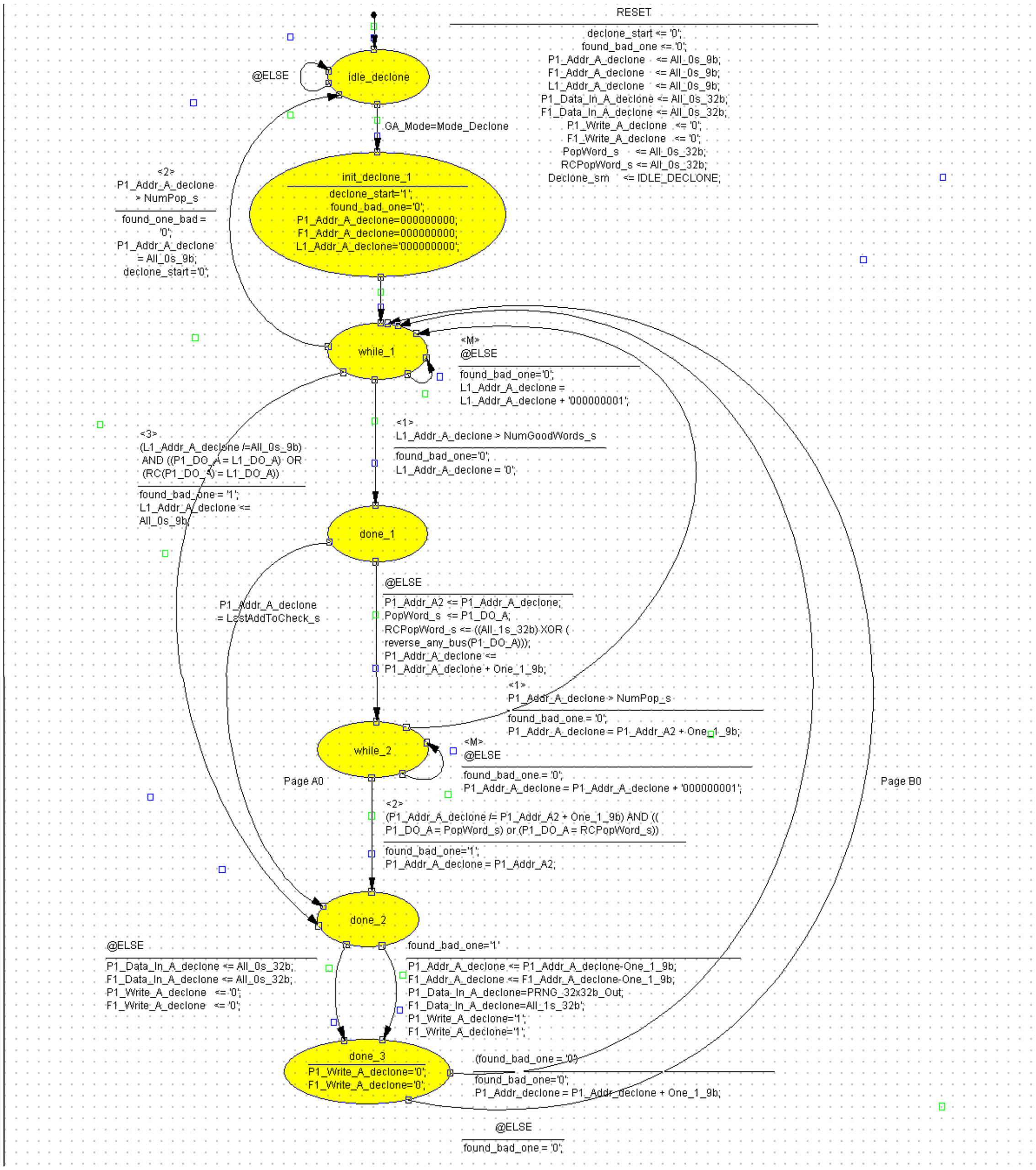
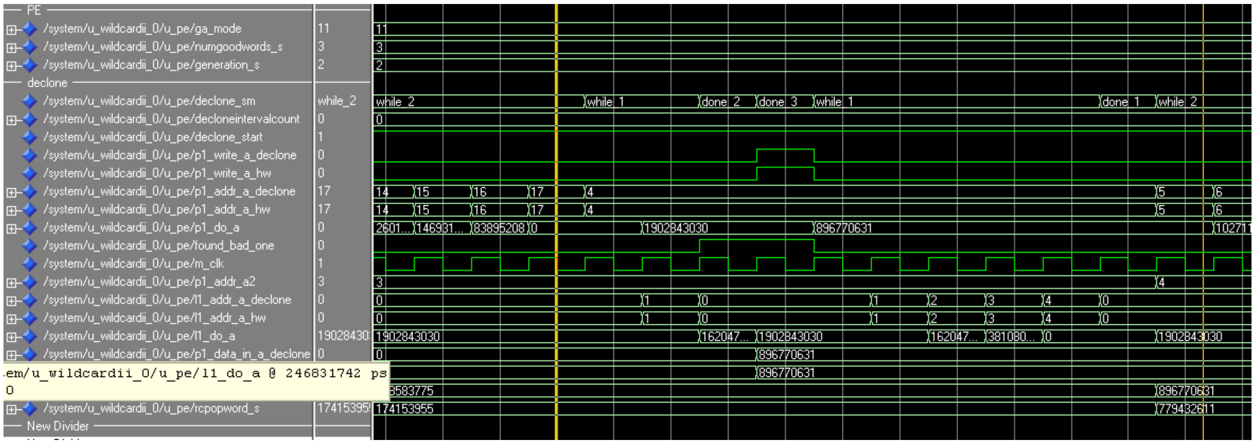

RESET
declone_start <= '0';
found_bad_one <= '0';
P1_Addr_A_declone <= All_0s_9b;
F1_Addr_A_declone <= All_0s_9b;
L1_Addr_A_declone <= All_0s_9b;
P1_Data_In_A_declone <= All_0s_32b;
F1_Data_In_A_declone <= All_0s_32b;
P1_Write_A_declone <= '0';
F1_Write_A_declone <= '0';
PopWord_s <= All_0s_32b;
RCPopWord_s <= All_0s_32b;
Declone_sm <= IDLE_DECLONE;
@ELSE
idle_declone
GA_Mode=Mode_Declone
init_declone_1
declone_start='1';
found_bad_one='0';
P1_Addr_A_declone=000000000;
F1_Addr_A_declone=000000000;
L1_Addr_A_declone='000000000';
<2>
P1_Addr_A_declone
> NumPop_s
found_one_bad =
'0';
P1_Addr_A_declone
= All_0s_9b;
declone_start ='0';
while_1
<M>
@ELSE
found_bad_one='0';
L1_Addr_A_declone =
L1_Addr_A_declone + '000000001';
<1>
L1_Addr_A_declone > NumGoodWords_s
found_bad_one='0';
L1_Addr_A_declone = '0';
<3>
(L1_Addr_A_declone /=All_0s_9b)
AND ((P1_DO_A = L1_DO_A) OR
(RC(P1_DO_A) = L1_DO_A))
found_bad_one = '1';
L1_Addr_A_declone <=
All_0s_9b;
done_1
@ELSE
P1_Addr_A2 <= P1_Addr_A_declone;
PopWord_s <= P1_DO_A;
RCPopWord_s <= ((All_1s_32b) XOR (
reverse_any_bus(P1_DO_A)));
P1_Addr_A_declone <=
P1_Addr_A_declone + One_1_9b;
P1_Addr_A_declone
= LastAddToCheck_s
<1>
P1_Addr_A_declone > NumPop_s
found_bad_one = '0';
P1_Addr_A_declone = P1_Addr_A2 + One_1_9b;
while_2
<M>
@ELSE
found_bad_one = '0';
P1_Addr_A_declone = P1_Addr_A_declone + '000000001';
Page A0
Page B0
<2>
(P1_Addr_A_declone /= P1_Addr_A2 + One_1_9b) AND ((
P1_DO_A = PopWord_s) or (P1_DO_A = RCPopWord_s))
found_bad_one='1';
P1_Addr_A_declone = P1_Addr_A2;
done_2
@ELSE
P1_Data_In_A_declone <= All_0s_32b;
F1_Data_In_A_declone <= All_0s_32b;
P1_Write_A_declone <= '0';
F1_Write_A_declone <= '0';
found_bad_one='1'
P1_Addr_A_declone <= P1_Addr_A_declone-One_1_9b;
F1_Addr_A_declone <= F1_Addr_A_declone-One_1_9b;
P1_Data_In_A_declone=PRNG_32x32b_Out;
F1_Data_In_A_declone=All_1s_32b';
P1_Write_A_declone='1';
F1_Write_A_declone='1';
done_3
P1_Write_A_declone='0';
F1_Write_A_declone='0';
(found_bad_one = '0')
found_bad_one='0';
P1_Addr_declone = P1_Addr_declone + One_1_9b;
@ELSE
found_bad_one = '0';

Figure 5. State diagram of declone operator (in Xilinx StateCad tool).

```
------------------------
declone_pop: PROCESS (global_reset, M_Clk)
BEGIN
 IF rising_edge (M_Clk) THEN
   IF (global_reset = '1') THEN -- OR terminate = '1') THEN
      declone_start        <= '0';
      found_bad_one        <= '0';
      F1_Addr_A_declone    <= All_0s_9b;
      L1_Addr_A_declone    <= All_0s_9b;
      F1_Data_In_A_declone <= All_0s_32b;
      P1_Data_In_A_declone <= All_0s_32b;
      F1_Write_A_declone   <= '0';
      P1_Write_A_declone   <= '0';
      PopWord_s            <= All_0s_32b;
      RCPopWord_s          <= All_0s_32b;
      Declone_sm           <= IDLE_DECLONE;

   ELSE
      CASE Declone_sm IS

        WHEN IDLE_DECLONE =>
          IF (GA_Mode /= Mode_Declone) THEN
             Declone_sm    <= IDLE_DECLONE;
          ELSE
             Declone_sm    <= INIT_DECLONE;
          END IF;

        WHEN INIT_DECLONE =>
          declone_start     <= '1';
          found_bad_one     <= '0';
          --P1_Addr_A_declone <= All_0s_9b; -- elsewhere
          --F1_Addr_A_declone <= All_0s_9b; -- elsewhere
          L1_Addr_A_declone <= All_0s_9b;
          Declone_sm        <= WHILE_1;

        WHEN WHILE_1 =>
          -- bail if the Lib Addr is past last word.
          -- here we have not delayed a clk after addr 0 was set
          -- before entering, so no check is done if Addr=0.
          -- DO lags Addr by a clock, this is what is checked
          -- for NumGoodWords=3 (lib words in Addrs 0,1,2)
          -- Addr DO
          -- 0    x (no check)
          -- 1    0
          -- 2    1
          -- 3    2
          -- 4    bails (no check)
          IF (L1_Addr_A_declone > NumGoodWords_s) THEN
            found_bad_one     <= '0';
            L1_Addr_A_declone <= All_0s_9b;
            Declone_sm        <= DONE_1;

          -- bail if the Pop Addr is past last word
          ELSE
           IF (P1_Addr_A_declone > NumPop_s) THEN
            found_bad_one <= '0';
            --P1_Addr_A_declone <= All_0s_9b; -- elsewhere
            --F1_Addr_A_declone <= All_0s_9b; -- elsewhere
            declone_start <= '0';
            Declone_sm        <= IDLE_DECLONE;
           ELSE
            -- check if Pop or RCPop word is already in Lib
            IF ((L1_Addr_A_declone /= All_0s_9b) AND
                ((P1_DO_A = L1_DO_A) OR
```

```
                     (((All_1s_32b) XOR
(reverse_any_bus(P1_DO_A))) = L1_DO_A)
                    )
                   ) THEN
                   -- yes - set flag and bail L1 addr loop
                   found_bad_one <= '1';
                   L1_Addr_A_declone <= All_0s_9b;
                   Declone_sm <= DONE_2;
                ELSE
                   -- no - check against next lib word
                   found_bad_one     <= '0';
                   L1_Addr_A_declone <= L1_Addr_A_declone +
One_1_9b;
                   Declone_sm        <= WHILE_1;
                END IF;
               END IF;
              END IF;

          WHEN DONE_1 =>
             … (code removed for report)

          WHEN WHILE_2 =>
             … (code removed for report)

          WHEN DONE_2 =>
             … (code removed for report)

          WHEN DONE_3 =>
             … (code removed for report)

          WHEN OTHERS =>
             declone_start     <= '0';
             Declone_sm    <= IDLE_DECLONE;

        END CASE;

    END IF;  -- if global reset
  END IF; -- if M_Clk
END PROCESS;
------------------------
P1_Addr_B_declone_process:  process (global_reset,M_Clk)
begin
  IF rising_edge (M_Clk) THEN
    IF (global_reset = '1') THEN -- OR terminate = '1') THEN
      P1_Addr_A_declone    <= All_0s_9b;
      F1_Addr_A_declone    <= All_0s_9b;
    ELSE
      IF (Declone_sm = INIT_DECLONE) THEN
         P1_Addr_A_declone <= All_0s_9b;
         F1_Addr_A_declone <= All_0s_9b;
      END IF;

      IF ((Declone_sm = WHILE_1) AND
          (P1_Addr_A_declone > Numpop_s)
         ) THEN
         P1_Addr_A_declone <= All_0s_9b;
         F1_Addr_A_declone <= All_0s_9b;
      END IF;

      … (code removed for report)

    END IF;  -- for reset
  END IF; -- for clock
END PROCESS;
-----------------------
```

Figure 6. Partial code fragment of the declone operator, hand coded from state diagram.



```
PE
/system/u_wildcardii_0/u_pe/ga_mode                 11
/system/u_wildcardii_0/u_pe/numgoodwords_s          3
/system/u_wildcardii_0/u_pe/generation_s            2
declone
/system/u_wildcardii_0/u_pe/declone_sm              while_2
/system/u_wildcardii_0/u_pe/decloneintervalcount    0
/system/u_wildcardii_0/u_pe/declone_start           1
/system/u_wildcardii_0/u_pe/p1_write_a_declone      0
/system/u_wildcardii_0/u_pe/p1_write_a_hw           0
/system/u_wildcardii_0/u_pe/p1_addr_a_declone       17
/system/u_wildcardii_0/u_pe/p1_addr_a_hw            17
/system/u_wildcardii_0/u_pe/p1_do_a                 0
/system/u_wildcardii_0/u_pe/found_bad_one           0
/system/u_wildcardii_0/u_pe/m_clk                   1
/system/u_wildcardii_0/u_pe/p1_addr_a2              3
/system/u_wildcardii_0/u_pe/l1_addr_a_declone       0
/system/u_wildcardii_0/u_pe/l1_addr_a_hw            0
/system/u_wildcardii_0/u_pe/l1_do_a                 19028430
/system/u_wildcardii_0/u_pe/p1_data_in_a_declone    0
.em/u_wildcardii_0/u_pe/l1_do_a @ 246831742 ps
0
/system/u_wildcardii_0/u_pe/rcpopword_s             17415395
New Divider
```

Figure 7. Test waveforms for the declone process, showing a clone found and replaced with new random individual (in ModelSim).

We note here that waveforms of the type shown in Figure 7 must be inspected for a variety of conditions in order to prove that the code operates correctly in each state with different inputs. These displays can also be used to count clock cycles and construct execution time models that can predict the time required to execute the operators with various GA parameter values, e.g. population size, library size, etc. Another alternative for building such models is to instrument the code with timers, or clock cycle counters, that can report back to the host program how much time is spent in each process, say over a call or a generation. Such a version is actually under development, and is mentioned again in Section 5 under future work.

Finally, we also note that the ISE tools could be used to check speed and resources for only portions of the design, but not the entire project, at least when targeting the WildCard and WildStar boards. This was because the vendor (Annapolis Microsystems) only provides libraries that support synthesis with the Synplify tool. The amount of work needed to modify the libraries to support the ISE tool would have been prohibitive, so we just used Synplify for synthesis.

2.3 Hardware Prototype Platforms

Four hardware platforms were used at various times by different people during this project, as shown in Table II. The WildCard and WildStar platforms were mainly used at RITC, while both of those and the XUP V2P platform were used by summer faculty and students.

Table II. Hardware prototype platform reconfigurable logic and on-chip memory resources.

FPGA board	Xilinx FPGA	Logic Cells	BRAMS (Kbits)	Embedded PPCs	Cost	Bus
XUP V2P	XC2VP30	30,816	2,448	2	$1.6K	various (standalone w/ cable to PC)
WildCard2	XC2V3000-4	28,672	1,728	0	$3.5K	PCMCIA (notebook)
WildCard4	XC4VSX35-10	34,560	3,456	0	$1.8K	PCMCIA (notebook)
WildStar	XC2VP70	74,448	5,904	2	$16K	PCI-X (workstation)

In terms of technology generation and potential speed, the oldest, and slowest chip is the XC2V Virtex 2 family device on the WildCard2, followed by the XC2VP Virtex-2 Pro chips on the XUP V2P and WildStar boards, and finally the relatively newer and faster XC4VSX Virtex 4 family device on the WildCard4. In general, we were able to obtain higher clock frequency estimates and actual post-synthesis actual clock speeds for the boards that used the later generation chips. It should be noted, however, that our clock speed result depended on a number of factors, such as subtle differences in the design versions we implemented from time to time on the different platforms, the stochastic nature of the synthesis tools, and the amount of effort requested of the synthesis tools by the operator at synthesis time. Generally we used similar project setup files and tried to keep these variables constant.

2.4 Prototype versions and design features

Table III shows information about each of a series of 4 versions of prototype designs that we implemented and tested during spiral 3. The first column indicates the version number and application name, the fitness metric, and major new features of the version. The remaining columns show how many copies of the PEs and the metric calculator were used in the versions, the types of GA operators, the hardware platform(s), and conferences where the designs were described and references.

Table III. Prototype version and design features.

(version) application name, fitness metric, new features	# PEs/fitness evaluators	selection/ mating	mutation	Hardware platform(s)	Publication [reference] (or in writing)
(1) GA/DNA Codes, RC LLCS metric, exhaustive search (ES)	1/1 GA 1/2 ES	n/a	best of 48	XUP V2P WildCard-II WildStar-II	GECCO 2006 [23] MAPLD 2006 [24]
(2) GA/DNA Codes, RC LLCS metric, multi-deme GA	2-5/1 GA 1/2 ES	random/ single point	best of 48	XUP V2P WildCard-II WildStar-II	CIBCB 2007 [25] (IEEE J. Computational Intelligence) [26] GECCO 2007 [27]
(3) GA/DNA Codes, RC Gibbs metric, thermodynamic constraints	1/1 GA	n/a	best of 48 or new random	WildStar-II	DNA 2007 [28] (Springer LNCS) [29]
(4) GA/DNA Codes, RC LLCS Codes, rank base selection and declone	1/1 GA 1/2 ES	rank based/ single point cross	best of 48 or new random	WildCard-II WildStar-II WildCard4	(GECCO 2008?) [30]

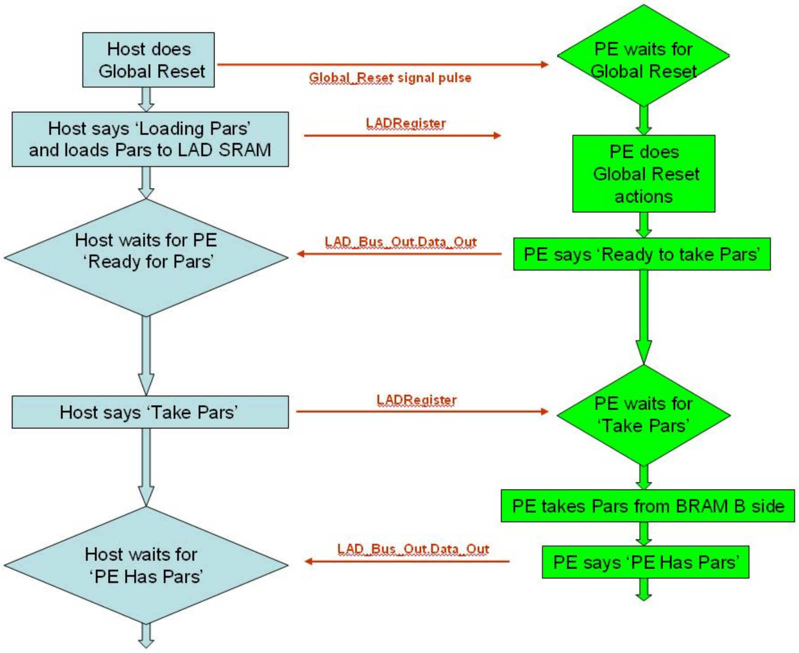
These versions shared many basic design features. For example, they each consisted of a C host program, and a PE image that was synthesized from a collection of VHDL process files. In general, the operator used them by invoking the compiled host program in a DOS window on the host PC workstation (or notebook). A number of command line arguments could be specified at run time for both the GA and DNA Code building parts of the application, as shown in Table IV.

Table IV. Command line arguments for GA/DNA FPGA application.

option <type>	description (range)	default
-a <int>	set # bases in code words (must be 16)	16
-d <int>	set device \"slot\" number	0
-e <int>	set exhaustive population checking flag (0/1)	0
-f <int>	set PE clock frequency (MHz) (10-300)	100
-g <int>	set max_gens (1-4,000,000)	10,000
-h	show this help	
-i <int>	set initial population size (10-511)	16
-k <int>	set # keepers (10-511)	16
-l <int>	set initialization type (0/1/2 easy/random/passed in)	1
-m <int>	set max_match (2-16)	10
-r <int>	set running population size (10-511)	16
-s <int>	set random_number_seed (0-max int)	1
-t <int>	set maximum run time (sec) (1-10,000)	60
-v	sets verbose mode to show progress messages	
-w <int>	set # code words to generate (20-300)	100
-x <int>	set mutation type (0)	0
-z <float>	set percent mutations (0-100)	1.0

The host program started by initializing its own data arrays in memory, downloading the PE image file to the PE (that defined its function). Then it interpreted the command line arguments and set up a block of integer parameters, passed them to the PE, and verified that the PE had received the parameters. Then it entered a main loop and received words back from the PE as they were found. At first, communication between host and PE was done on the Annapolis boards using example programs supplied by the vendor, but later a new interface was written that provided for double sampling across the clock boundary between the interface bus and PE bus, which ran at different frequencies. This approach seemed to cure what appeared to be intermittent communication noise. We used a handshaking protocol to keep the host and PE in synch during communication events between host and PE. An example of part of the communication flow at startup is shown in Figure 8. The PE reported words back to the host as they were found. The host kept track of elapsed run time, time stamped the words as they were found, and stored them in memory and in disk files for later analysis. The host program also controlled running a sequence of tests, and produced curves of words found vs. time for each run and averaged over multiple runs. It also tracked the total number of words found by the GA, and optionally by exhaustive search for each run and averaged over all runs.

Figure 9 shows a simplified program flow chart for the PE. It began by initializing the DNA Code word library to either empty or to an initial set of words that could be passed in from the host. It also initialized the GA population to the specified size, and checked the fitness of the individuals in the initial population. Then it entered a main loop that found words until the run was terminated by one of three criteria (maximum elapsed time, maximum number of generations, or the desired number of words were found). Each pass through the loop defined a generation, including a population checking step, and a mutation and checking step, and finally a step that produced a population for next generation by applying sorting, selection, and mating operators. New words were picked up and added to the library during each of the two checking phases if their fitness was such that they satisfied the constraints required to enter the library. The following sections describe the features of each version that was designed and tested.



Host does Global Reset
Global_Reset signal pulse
PE waits for Global Reset
Host says 'Loading Pars' and loads Pars to LAD SRAM
LADRegister
PE does Global Reset actions
Host waits for PE 'Ready for Pars'
LAD_Bus_Out.Data_Out
PE says 'Ready to take Pars'
Host says 'Take Pars'
LADRegister
PE waits for 'Take Pars'
PE takes Pars from BRAM B side
Host waits for 'PE Has Pars'
LAD_Bus_Out.Data_Out
PE says 'PE Has Pars'

Figure 8. Host/PE communication handshaking at startup.

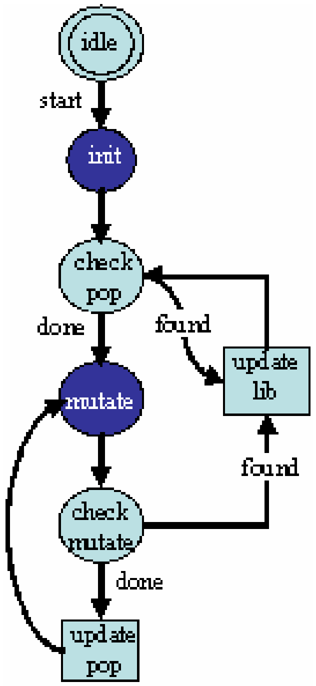


Figure 9. PE flow chart.

2.4.1 Version 1: GA/DNA Codes with exhaustive search

Prototype version 1 implemented a first cut that we thought would be a maximally fast version of the application. It used no selection and mating operators at all in the GA, because we had observed during tests of the software versions in spiral 2 that mating did not always help speed the discovery of codewords. It also used a unique, accelerated mutation operator that selected individuals at random for mutation, as would typically be done, but which then checked every possible mutation of the 16 mer that could be made. This was attractive for two reasons. First, it avoids what would be considerable delay and overhead associated with randomly selecting the same individual over and over again to generate those mutations. Second, it makes better use of the systolic array fitness metric calculator by checking a group of mutations and spreading the checker's pipeline latency of 15 clocks over 47 fitness calculations instead of 1 calculation. This is important during the early part of the run when the number of words in the library is small. For example, with n words in the library, the utilization factor of the systolic array (calculations/clocks) is about n*47/(n*47+15), which for n=1 is 0.76. If we checked each of the 47 mutations by itself, the utilization factor would be n/(n+15), which for n=1 is 0.062. Actually, further improvement ns speed could have been made by streaming sets of mutations for multiple individuals together through the calculator, e.g. for 2 individuals utilization would reach 0.85, for 3 individuals 0.9, and for 10 individuals 0.97. But that improvement was not pursued because n grows rapidly as words are found, and the utilization factor quickly approaches 1.0, e.g. at n=25 it is 0.987, and at n=50 it is 0.994. Another consideration for the mutation operator is that it may or may not identify a mutation that improves fitness compared to the original individual. If it does, the best of the 47 tried is used to replace the original individual in the population. If it does not, there are several options, including using the original individual, picking one of the 47 a mutations randomly, or replacing the individual with a new random individual. Most of the early versions used the second option, but later versions optionally used either the second or third (which typically worked better). Finally, GA/DNA version 1 used about 42% of the WildCard-II FPGA chip resources, and about 16% of the WildStar card FPGA resources, and both ran at 100MHz.

Another feature of this version was the capability to do exhaustive search (ES) to extend an existing library. This involved checking every word in the universe of 2^(32-1) = 4.3E9 possible words against the library, after the library was initially built by running GA. This has hitherto been impractical for codes with word length 16, because it would take an estimated 62 days on a 2GHz Pentium workstation. This hardware version does this checking in only 1.5 hours running at 100MHz. The ES capability actually was implemented as a separately synthesized version of the PE, but it is described here since it was developed along with version 1. The PE image for ES was loaded and run under control of the host program, typically after an initial library was composed by running GA/DNA version 1 for a chosen amount of time, eg. typically about 10 minutes. The ES version finds all the remaining words that can possibly be added to the library by using a 32 bit counter (rather than the GA population) that starts at 0 and sequences though all possible candidate words, checking each against the library. When a new word is found, it simply adds it to the library, and the search continues. The ES version actually used 2 LLCS systolic arrays to do the checking, one to process the candidate words, and another to process the reverse complements of the candidate words.

While ES does not guarantee finding the *global optimum* sized library because that would require doing a run with each possible sequence of random numbers, which is impossible due to time constraints. However, it does guarantee finding what we call a *locally optimum* library, in the sense of the 2^(32-1) possible 32 bit 16 mers have been checked for possible addition to the library. We can now easily run multiple runs with different random number seeds to generate many locally optimum libraries, and for the first time we have gathered statistics on the average size of libraries. This approach may actually yield better estimates of the upper bound on library size than present theoretical methods. Finally, the ES version with 2 fitness checkers uses about 75% of the WildCard-II resources, and <40% of the WildStar board FPGA resources, and both run at 100MHz.

The capabilities of the hardware GA/DNA Code and ES prototypes described here are thought to be novel and unique, and they were described in papers at an evolutionary computing conference [23], and at an embedded hardware conference [24].

2.4.2 Version 2: GA/DNA Codes with multi-deme GA

The second prototype version added the capability to run multiple populations on the same single FPGA chip, effectively creating a multiple node distributed Island Model GA on a single chip. This was motivated by our observation in spiral 1 that a multi-deme GA could provide about linear speed-up vs. the number of demes, or populations. It was actually possible to instantiate 2 PEs on the WildCard-II FPGA chip, and 5 PEs on the WildStar card FPGA. These designs also include an arbiter to handle communication between the multiple instances of the PE on the FPGA chip, as shown in Figure 10. Periodically each of the PEs send a migration request to the arbiter, typically after a set number (or epoch number) of generations. The arbiter acknowledges a request if its migration controller is in the idle state. After receiving the acknowledgement from the arbiter, the PE sends its best few individuals and their fitness values to the arbiter. These data are placed in a memory together with similar data received from other PEs. The arbiter sorts and picks the best *m* individuals, where *m* is the number of individuals to be migrated, and sends them back to the PE which initiated the request for migration. For the case of 2 PEs on a chip served by one arbiter, this is equivalent to a directed ring configuration. However, for the case of more than 2 PEs on a chip, this approach implements a star, or local pooling configuration. Above the chip level, the host PC is free to implement any communication configuration among multiple host nodes in a cluster, e.g. with standard MPI. Thus, multiple multi-deme GA chips could be used in a hierarchical system or a ring of host workstations.

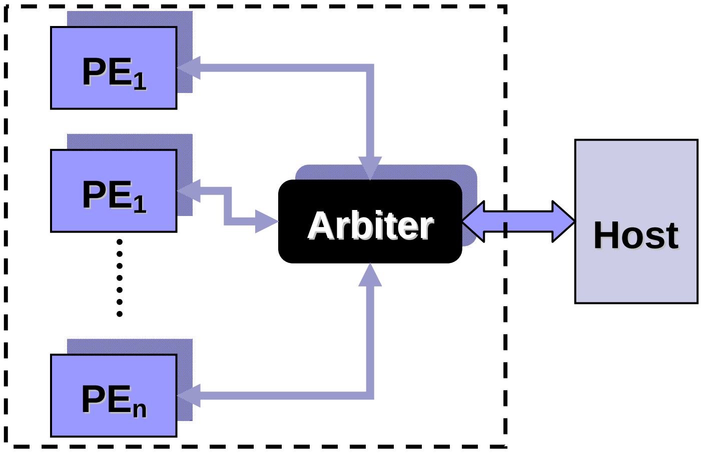


Figure 10. Multiple copies of PE and an arbiter on single FPGA chip.

Sorting, selection, and mating operators were also added to the GA for this version. In order to minimize resource utilization and maximize speed, very simple versions of these operators were used. Sorting was implemented by scanning the population repeatedly up to k times in order to identify and pick up, or keep, the k best individuals from the p population individuals. Typically, p ranged from 16 to 512, and k was some fraction of p, perhaps 1/8, although we could vary both. The selection method randomly picked two of the k individuals kept by the sort. Then a base index (0-15) was chosen, and the two parents were cut at this base boundary into a head and tail. Single point crossover was used to produce two children from the heads and tails of the two parents (head1+tail2 and head2 + tail1), and the children were added to the kept list. This was repeated until the new population grew from k to p. We note that this selection method lacks the selection pressure of rank or fitness proportional probabilistic selection (which was done later in version 4), but it was fast. Following mating, mutation was done by the same method described in 2.3.1 for version 1, until a chosen number of mutations had occurred. This version actually had a couple of minor design bugs that should have been fixed, but we don’t think they seriously degraded performance. One was that when a word was picked up into the library it was left in the population, rather than being replaced by a new random word. Another was that the mutated word address was generated as a 9 bit random number (0-511), meaning that addresses up to 511 were used, even though the population p could be 4 to 511. Since only words 0 to (p-1) were sorted, checked and searched for pickup, mutation operations were wasted for if p was < 511. Both of these bugs were fixed in later versions. Finally, the multi-deme GA/DNA application used almost all of the resources of the FPGA chips, with 2 PEs instantiated on the WildCard-II board, and 5 PEs instantiated on the WildStar board. The same 100MHz exhaustive search PE described previously was used to extend codes generated by this version, which again took 42% or 16% of the 2 cards' resources.

We believe that this hardware multi-deme GA, and the hardware DNA Code building application are both novel and unique at this writing. The have been described in papers at a bioinformatics conference [25] and journal [26], and an evolutionary computing conference [27].

2.4.3 Version 3: GA/DNA Codes with thermodynamic constraints

The version 3 prototype was similar to the version 1 prototype, except that it used a single PE, and a single population GA, but it substituted an improved hybridization fitness metric called the Gibbs free energy metric for the LLCS metric. The Gibbs metric calculates an estimate of the binding energy of two mers using a nearest neighbor model that considers the specific contiguous two-by-two sets of base pairs (2 stems) that occur along the mers, again in all alignments of the mers. In software, this requires the calculation of 3 matrices instead of 1 for the LLCS, and the hardware implementation takes significantly more resources than LLCS. The original software dynamic programming algorithm for calculating Gibbs energy requires access to possibly distant cells along the diagonal to the upper left of the cell being calculated. This is problematic in a hardware implementation of a 2D systolic array, not only because of wiring requirements, but also because at the time this 'look back' needs to be done, the cells that calculated the data are already busy calculating on other operands, and the data is lost. One possible solution would be to add multiple registers (or use memory) to store the needed data. However, a much better solution was found that involves modifying the equations to also calculate a *running minimum* quantity in each hardware cell (in 2 of the 3 arrays), which are then passed forward to adjacent cells. This satisfied the data locality requirement for building a pipelined 2D systolic array, i.e. all the data necessary to calculate the outputs of each cell are available at the inputs of the cell (rather than on wires coming from non-adjacent cells).

The constraints in the fitness function were also modified for this version. In the previous versions that used the LLCS metric, in order to enter the library a candidate word (and its reverse complement) must have a certain minimum edit distance when compared to its own reverse complement, and when compared to all of the words already in the library (and their reverse complements). Using the Gibbs energy metric, in order to enter the library the Gibbs energies of all potential unintended cross-hybridizations (e.g. measured between each candidate word and its reverse complement, and each library word and their reverse complements) had to fall below a certain threshold and within a certain range. The threshold and range could be adjusted to ensure that binding energies of unintended cross-hybridizations was poor (had suitable low melting temperatures) compared to the energies of intended hybridizations within Watson Crick pairs in the library (had higher melting temperatures).

This version also added 2 new options for the mutation operator. One was to simply replace the individual chosen for mutation with a new random individual, instead of generation of the 47 possible mutations. The other was similar to the previously used mutation operator, i.e. all 47 possible mutations were tried, but when no better word was found, either one of the mutations could be selected at random, or a new random individual could be inserted in place of the original word. The pickup procedure was also modified to replace a word that was picked up and put into the library with a new random word, rather than leaving it is the population. An additional feature was added to a later variation of this version that optionally used a counter instead of a random number generator to source new word values to be inserted in the population, e.g. when a word was picked up into the library, or when a mutation did not result in an improvement. This was done to enable comparing *deterministic* versions of the hardware and software algorithms, i.e. ones that used identical sequences of 'random numbers' in both software and hardware. The PE for version 3 would not fit into either of the WildCard boards, but it did fit on the WildStar board. It used about 92% of the WildStar card's resources, and operated at 100MHz. Also, an exhaustive search version of this PE was done, and it used 90% of the WildStar resources, and ran at 100MHz.

Finally, we believe that this version's hardware implementation of the Gibbs metric in a systolic array covering all alignments of short mers is novel and unique at this writing, and it has been described in a paper at a conference that focuses on bio-molecular computing [28], and in a book series that published expanded versions of invited papers from the conference [29].

2.4.4 Version 4: GA/DNA Codes with rank based selection and declone

The last hardware version described here, version 4, was a modification of version 1 that added an improved selection and mating operator, and a decloning procedure for removing clones from the population every few of generations. This version used the LLCS metric, 1 PE, and also fixed the 2 bugs mentioned previously in version 1. The new selection method used rank based selection of parents for mating. First the population is partially sorted according to fitness to identify the top k individuals that are kept. Then, parents are chosen from the k individuals with probabilities that are proportional to their rank. Thus, better individuals are chosen more often to act as parents. Mating is then done using single point crossover, as before, until the population grows back to the original size. Finally, mutation is done using the best of 47 method, with a new random individual inserted if none of the mutations were better.

One potential difficulty when using a GA is a lack of diversity after many generations, e.g. the population may develop multiple copies of the same individual, especially if the mutation rate is too low. This may collapse the search to a local minimum. However, this problem can be fixed by detecting this situation and doing a restart, or by periodically introducing many new random individuals, or by running a decloning procedure that replaces each clone with a new random individual. We chose to implement a decloning procedure, but since this process costs time (of order p*c, where p is the population size and c is the number of clones), we added a declone interval counter to enable running the declone procedure only every few generations. The decloning procedure also checked for and removed words in the population that were duplicates of words (or their reverse complements) already in the library (or their reverse complements). Such words have fairly good fitness, because they are rejected only by the duplicate word (and its reverse complement) that are already in the library, but their fitness will never improve. Finally, this version used about 47% of the resources of the Wildcard4 board FPGA, and it ran at 100MHz.

Version 4 has been synthesized and tested, but not described in a paper at this writing. It would be appropriate to describe this version and compare the results with that of the other versions (especially versions 1 and 2) at an evolutionary computing conference [30] . In fact, we have proposed to the organizers of GECCO 2008 to do a workshop that would focus on hardware implementations of EC algorithms and applications, and received a positive response.

We also note that there was some work done during this spiral on other features of some of these versions that is not reported here, because it was related to future work. For example, we did design a 32 x 32 mer version of the LLCS systolic array, and synthesized it to determine the impact on size, and found that it took 50,686 LUTS, compared to 12,827 LUTS for the 16 x 16 mer array. This is slightly less that 4 times the resources, and is slightly better than what might be expected. This version would not fit on the WildCard boards, but would fit onto the WildStar board.

3.0 Results - spiral 3 (hybrid software/hardware accelerated platform)

This section summarizes the major findings of spiral 3 and then reviews some specific test results for each of the 4 main prototypes.

3.1 Major Findings

These hardware accelerated prototypes all represent the first ever full integrations of a genetic algorithm and an exhaustive search capability for generating non-cross-hybridizing DNA Codes up to word length 16. They also achieved extreme speed-ups on the order of 1000X that enabled us to do extensive multiple run statistical tests that resulted in a number of first time observations. For example, using prototype version 1, we determined that running a simplified GA with a modified locally exhaustive mutation only operator for 5-10 minutes routinely finds over 98% of the total number of codewords that can be found (by extending the code using our hardware exhaustive search prototype). This result was completely unknown in the literature prior to this work, mainly due to the fact that exhaustive search of words of length 16 was impractical. We also observed typical code sizes of about 122 word pairs for length 16, distance 6, reverse complement LLCS codes, and we think that this actually represents a higher upper bound than that predicted by existing theoretical methods. The second prototype represents what we think is the first ever implementation of a hardware multi-deme GA, and the first integrated with any type of non-trivial hardware application problem. It demonstrated the expected 2X speed-up for a 2 node version, again on the difficult DNA Code problem. This version has paved the way for hierarchical implementations that could use a cluster of workstations hosting such FPGA chips. The third prototype represents what we believe is the first ever hardware implementation of a Gibbs free energy of binding metric calculator for short DNA strands. The testing done with this prototype also established that it ran at 45MHz on a XC2VP70 FPGA chip, achieving about a 260x speed-up over a compiled C software version running on a Dell Precision 670 workstation with 3 GHz Pentium 4 processor. The fourth prototype demonstrated that a full GA, i.e. one with typical operators such as rank base selection, single point crossover, single point mutations, and population decloning took about 47% of LUT resources, and less that 10% of the on-chip block RAMs of even the smallest FPGA chip available to us, the Wildcard-II's XC2V3000 chip. This suggests that a variety of other problem types that require even more complex fitness function calculators and more memory could be implemented in hardware hosted on a notebook computer as well.

3.2 Prototype Test Results

This section presents test data illustrating the main results obtained with each of the prototype versions. More details on each can be found in references shown in Table III.

3.2.1 Version 1: GA/DNA with exhaustive search

Figure 11 shows the main results obtained by tests of this version [24]. Again, it utilized a minimal GA that did only a simple, modified mutation operator that selected an individual for mutation, and quickly checked all 47 possible mutations of the individual. This figure shows the performance of the algorithm in terms of the time taken to discover words as the library fills, where lower curves indicate faster (better) performance. The upper curve in Figure 11 is for a non-GA algorithm that used a Markov process that was among the best found in the literature at the time [31].

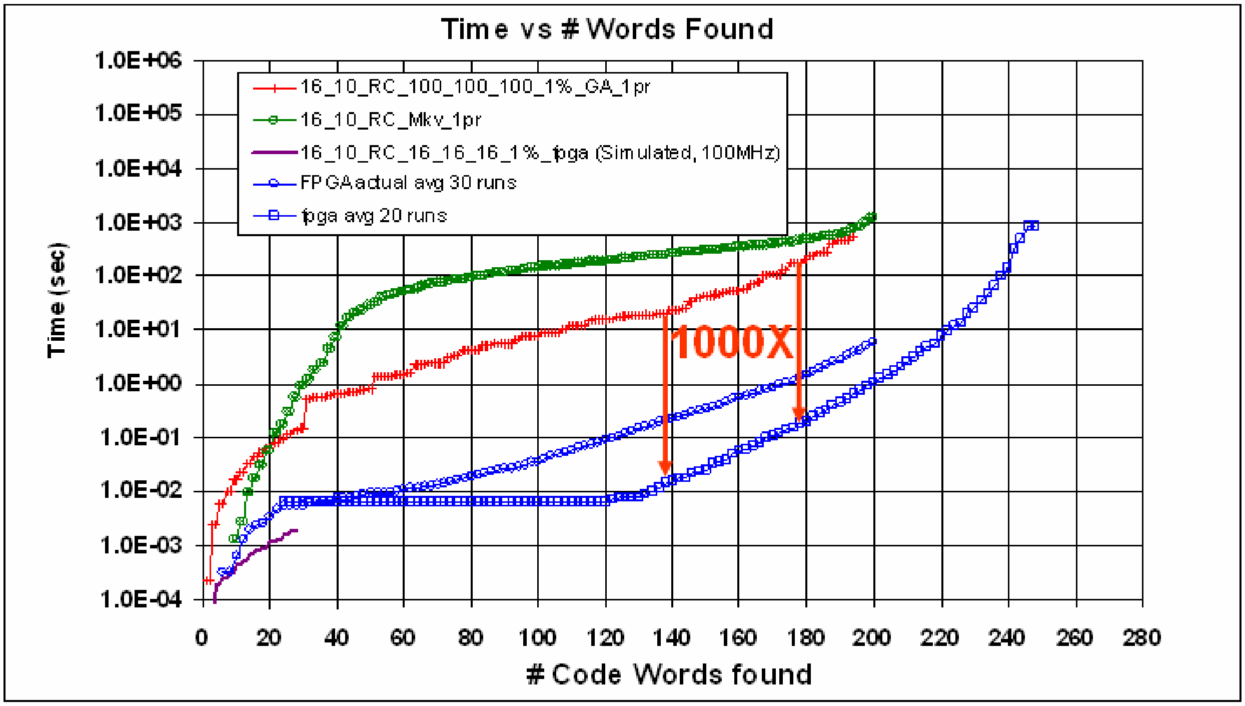


Figure 11. DNA Code generation application average performance using various optimization algorithms. Top: Markov guided; second from top: software GA; third from top: hardware GA run at 30MHz; third from top: hardware GA run at 100MHz; lower left: simulated hardware GA.

Note that the software GA found words somewhat faster that Markov toward the middle of the curve, and that the curves for both algorithms turn up at about 200 words (about 90% of the words that will be found), as the constraints posed by having so many words in the library make it difficult to find more words (the remaining 10%). Comparing the curves for software GA and the lower hardware GA curve shows that this version achieved about a 1000X speed-up. We note that the software GA was run on a 2.4GHz Pentium 4 workstation, and the hardware GA was run on a 100MHz FPGA. We also note that further improvements could be had by transferring the hardware to either a latest generation FPGA (~450MHz) or to an Application Specific Integrated Circuit (ASIC), which would be expected to achieve about 500MHz – 1.0 GHz clock speeds.

Figures 12 and 13 shows results obtained by composing initial libraries by running GA for 10 minutes, and then extending the codes by running exhaustive search (which takes an additional 1.5 hours) [25]. Figure 12 shows the total number of words found by GA and ES for a series of 90 runs generating length 16, distance 6 RC LLCS codes. On average, 120.4 words were found by GA, and 121.7 were found by GA + ES, which means that GA alone on average found 98.9% of the word pairs that can be found. Figure 13 shows a histogram of the number of runs vs. the number of word pairs that are added by ES, for a series of 32 runs. It shows that in about 1/3 of the runs GA found all the word pairs that can be found, for about 2/3's of the runs GA found all but 1 or 2 word pairs, and that for 3 runs ES added 4 word pairs.

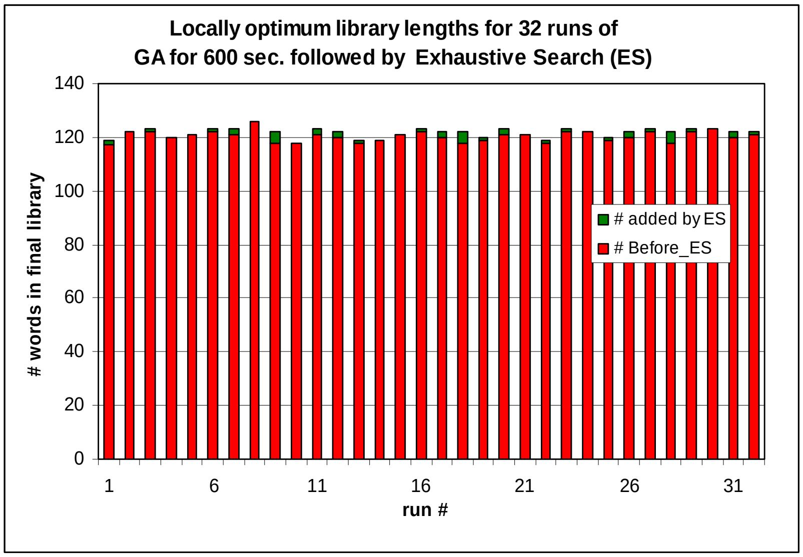


Figure 12. Sizes of libraries built with 10 min. of GA followed by exhaustive search.

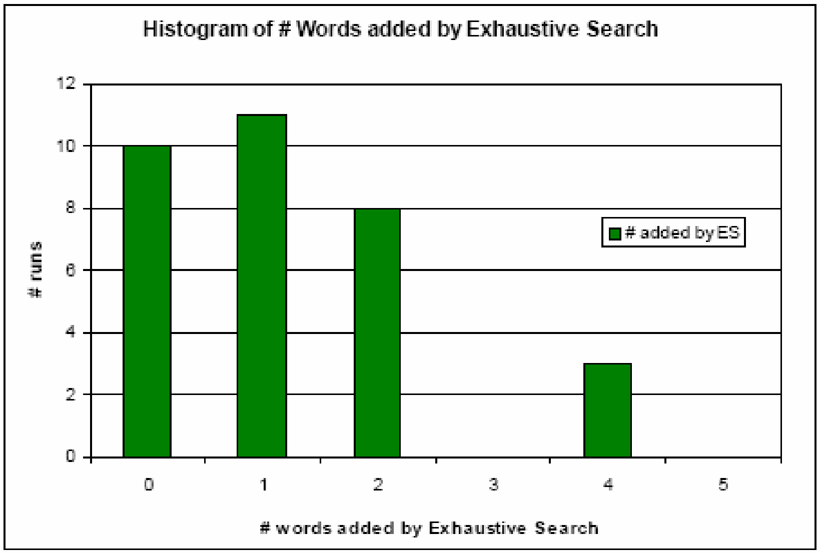


Figure 13. Histogram of number of words added by exhaustive search for the runs of Figure 12.

At this writing we do not have data for generating such codes with any other algorithm, because to date these types of experiments have been computationally impractical. For example, without the 1000X speed-up enabled by the hardware ES, 32 runs of GA in software would have taken about 1000 x 32 x 10 min. = 320,000 min. = 7.4 months, and 32 runs of ES in software would have taken 1000 x 32 runs x 1.5 hours/run = 48,000 hours, or about 5.5 years. With this prototype they took about 2 days. While it is true that a 1,000 node cluster could also have run these tests in 2 days in software, we know of no one who has done it, Further, the FPGA platform costs only $1.8K, and used only 1 notebook computer.

3.2.2 Version 2: GA/DNA with multi-deme GA

Figure 14 shows test results obtained with prototype version 2, which incorporated a multi-deme GA [27]. As expected, it shows approximately a 2X speed-up using 2 PEs vs. using 1 PE. Both curves are without mating and migration.

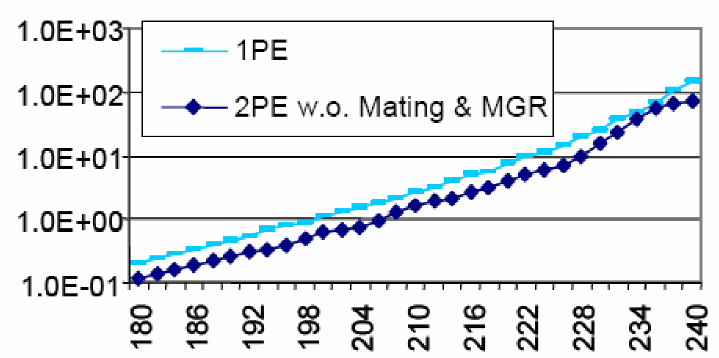


Figure 14. Performance comparison of prototype version 2, using 1 and 2 PEs.

Also from [27], Figure 15 shows an interesting effect that helps make a point about moving from software to hardware implementations. It shows the performance of this version with different % mutation parameter values. This parameter effectively controls the number of times per generation that the algorithm selects individuals for mutation during execution of the mutation operator. Indirectly, it also effects how often the mating and migration operators are performed, i.e. fewer mating operations take place in a given amount of time if the % mutation is larger. This is because more candidates are checked per generation during mutation, and less time is spent doing mating and migration. What is interesting is that in a software implementation such an effect, though present, would probably not even be noticed because the time spent on algorithm overhead is so small compared to fitness checking (2% in our case). This highlights the importance of thinking about what happens in the optimization algorithm when fitness evaluation time shrinks. Factors that effect the algorithm overhead time become important, and may impact the choice of operators and their parameters. It does not always follow that what works well in software predicts what will work well in hardware, and this may require some re-thinking when crafting hardware GAs.

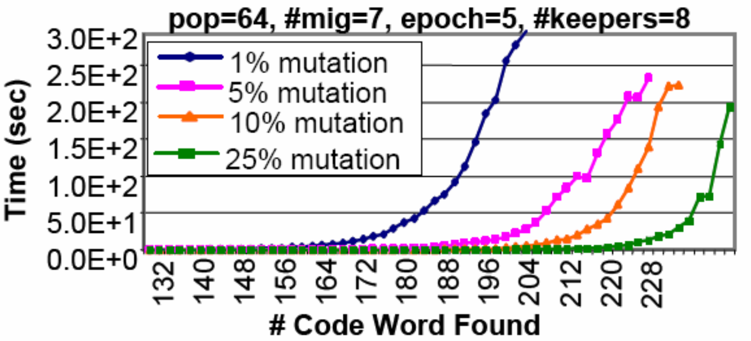


Figure 15. Effect of % mutation on performance, prototype version 2.

To carry this thought further, Figure 16 shows the effect of population size on performance, again from [27] with this multi-deme GA prototype. What we see is that again smaller population sizes are more efficient in terms of finding word faster, and again the reason is that algorithm overhead execution time has become important. Specifically, the time it takes to do the sort in the mating step (to identify the 8 keepers, which is the same for all the curves) grows linearly with population size, and so smaller population sizes have an advantage. Working in software the tendency may be to think that larger population sizes are better for diversity and performance, and they well may be if algorithm time is dwarfed by fitness function evaluation time. But in this case we see just the opposite working in hardware. A couple of comments are in order here, however, to explain why we do not see a clear linear relationship in the performance change vs. population size. One is that there is a confounding factor in this experiment, i.e. the selection pressure is changing because the number of keepers was constant across the tests. Thus, there was more selection pressure for the higher population sizes. Although this would be expected to improve performance, here it does not. Another is that we usually did not keep track of the number of generations, but do have some evidence that clones appear in the populations eventually, and this can hurt performance.

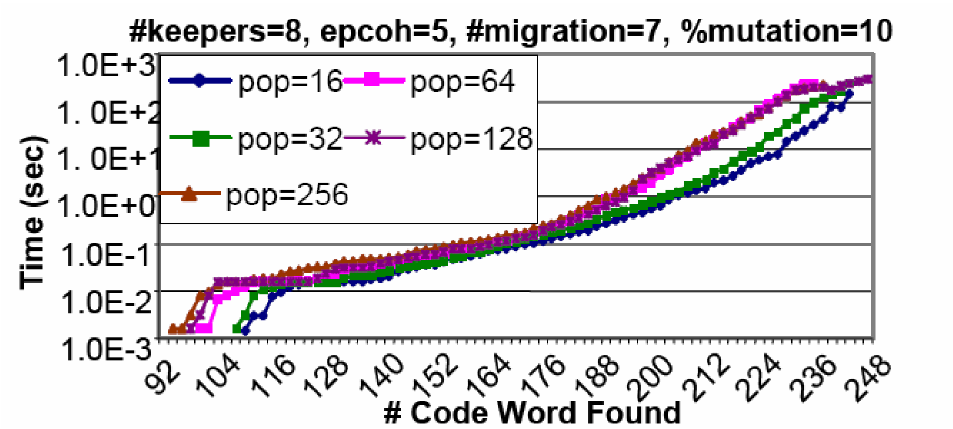


Figure 16. Performance of version 2 for different population sizes.

3.2.3 Version 3: GA/DNA with thermodynamic constraints

Figure 17 compares the performance of software and hardware versions of the third prototype that used thermodynamic constraints on cross-hybridization to compose 16 mer DNA Codes, i.e. using the Gibbs free energy metric instead of the LLCS metric. For details on what this metric is, and how it is implemented in hardware, the reader is referred to [28]. Here we just say that this metric is preferred by many researchers as being superior to the LLCS metric. Although it is more complex than LLCS, and requires fixed point calculations, it can be implemented in 3 systolic arrays that fit into the FPGA on the WildStar platform. The systolic arrays effectively calculate the metric for all alignments of the 2 mers. The top 2 curves in Figure 17 are for software versions, and the lower 2 curves are for hardware versions. The hardware versions achieved about a 260X performance speed-up over the software versions.

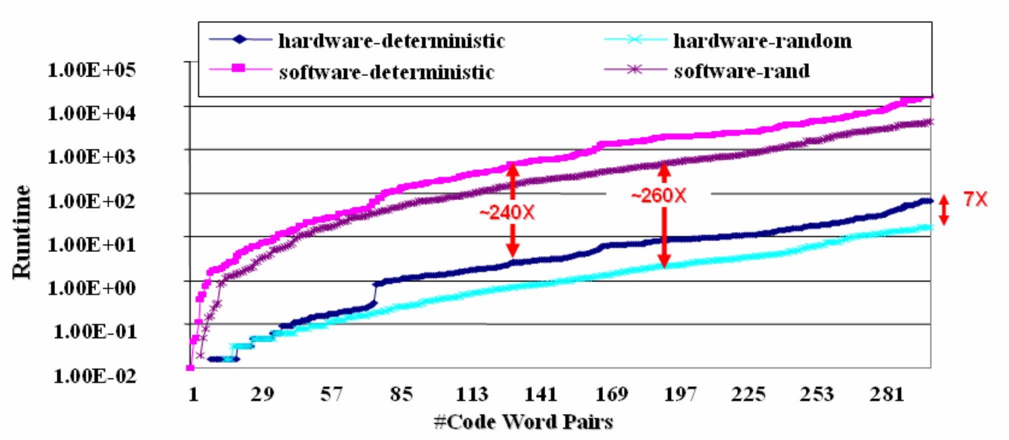


Figure 17. Performance of software and hardware version of prototype 3 (Gibbs energy metric).

There are 2 software and 2 hardware curves in Figure 17 because two variations of each were written in order to address a concern that exists when characterizing the performance of stochastic algorithms such as the GA. The concern is that each time the algorithm is run, the pseudo random number generator must be initially seeded with a random number. This is typically accomplished by keying the seed to the absolute clock time. As the GA runs it samples random numbers from the generator only occasionally, and the results of the run depend on exactly what set of random numbers are sampled. It is almost impossible to drive the software and hardware versions with exactly the same set of random numbers, even if they are initialized with the same seed. This is because they use different implementations of pseudo random number generators. Therefore, the software and hardware versions do not find the same set of words, and hence are not doing exactly the same calculations. We typically can only make statistical observations about how one version compares to another, e.g. by running a number of tests at different times, with different random number seeds, and averaging the results. This approach takes time. However, one way to overcome this problem is to implement exactly the same number generator in both versions, and seed them the same. This should result in exactly the same sequence of random numbers being generated in both versions, they should do the same calculations and find the same words. However, it would be hard to implement the random number generator used in the software version in hardware, and it would be slow to implement the hardware version's generator in software. Therefore, a compromise is to use something a lot simpler in place of the random number generator, e.g. a counter. While it is known that this negatively impacts exploration of a large search space, it does in theory guarantee that both algorithms will do exactly the same calculations. This means that speed comparisons can be based on only 1 run of each version instead of 20-30 runs. However, we note that in order to determine something like the average library length that can be found, one still has to do statistical test of multiple runs with different random number sequences (preferably with a pseudo random number generator, not a counter). At any rate, it is interesting to note that the GA still works quite well when driven by a counter, but it does find words about 4 times faster when driven by a pseudo random number generator.

The constraints on Gibbs energy for this version were set by specifying values for two parameters, a threshold, and a range. Together these parameters determine the allowable ranges of Gibbs energies for intended and unintended cross hybridizations in the library. A number of experiments were done to characterize the performance for different threshold and range parameter settings. An example of the results is shown in Figure 18 for an experiment that ran GA until no further words were found for 10 minutes, and then extending the resulting libraries with exhaustive search [28]. This was repeated for multiple values of the range parameter, with the threshold parameter fixed.

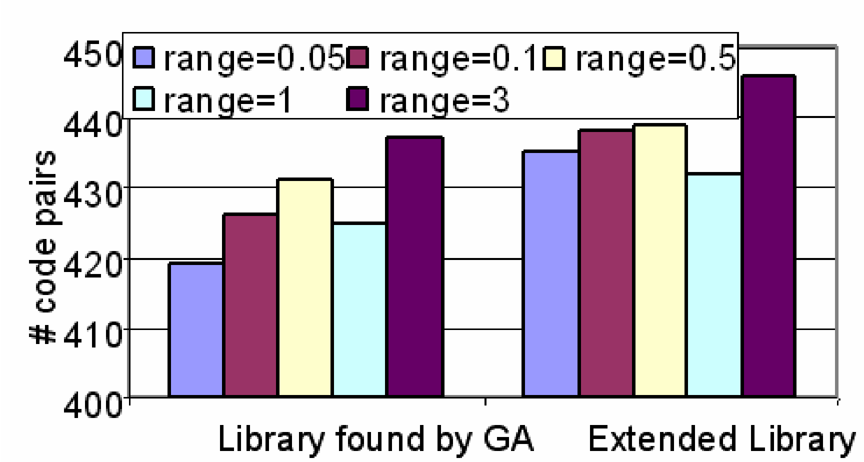


Figure 18. Library sizes found by GA and exhaustive search for prototype version 3 using thermodynamic constraints (fixed threshold, multiple range values).

We note two things about the results of this experiment. First, it becomes slightly harder to find words when the allowed range is smaller, which might be expected. Second, exhaustive search adds about 10-20 words to the 420-440 word libraries that GA found, which means that GA has found between 97% – 99 % of the words that can be found. The very good performance of the GA alone is similar to our observations with the previous prototypes.

The significance of the hardware accelerator is that it enables us to evaluate different code word search algorithms and explore the upper bound on the size of code word libraries in a reasonable amount of time. For example, without the hardware accelerator, each experiment in the second set would have taken more than 20 days. The total amount of testing done with this prototype that was reported in [28] is estimated to have taken about 16 hours in hardware, but would have taken almost 6 months if done in software.

3.2.4 Version 4: GA/DNA with rank based selection and declone

This final version incorporated the LLCS metric, a single population PE, rank based selection, single point crossover, and periodic decloning. Also, fixes were inserted for the previous bug involving best of 47 mutation when none increased fitness (the individual was replaced with a new random individual), for the word picked up from the population (they were replaced with new random individuals), and for restricting the addresses of mutation trials (to the maximum number of individuals in the population). Even though we had previously observed that mating did not significantly improve performance of the DNA code building application working with software versions in spiral 2, we implemented these operators in this version because they are generally used in many applications of GAs found in the literature. We thought this would increase the likelihood that our work could be transitioned to other problems with minimal work. We note here that our version 2 prototype did include mating and migration, but in the interest of design simplicity, the parent selection method was random, i.e. parents were chosen with uniform probability from the number kept by the sort. Selecting with uniform probability from the fraction of kept individuals does apply some selection pressure toward good individuals, and that pressure can be increased by keeping few individuals from which to select parents for the next generation. However, methods such as rank based selection can be used to apply even more selection pressure. This 4th version used a common method called rank base selection that selects parents according to a list of graded probabilities calculated from their rank, i.e. their position in a list sorted by fitness. This causes more fit individuals to be chosen more often as parents. It is known that this can be good and bad, good in the sense that better individuals might be bred faster in the population, but bad in the sense that the population may prematurely converge to a local optimum because it fills with clones. Many researchers try a number of different things to tune the performance of a GA to a particular problem, including varying the population size, the number kept from generation to generation (also called the crossover rate, or fraction kept), and variation of the selection, mating, and mutation operators.

Figure 19 shows the results of tests done with this version that looked again at the effect of population size on performance, as we did for some of the previous versions.

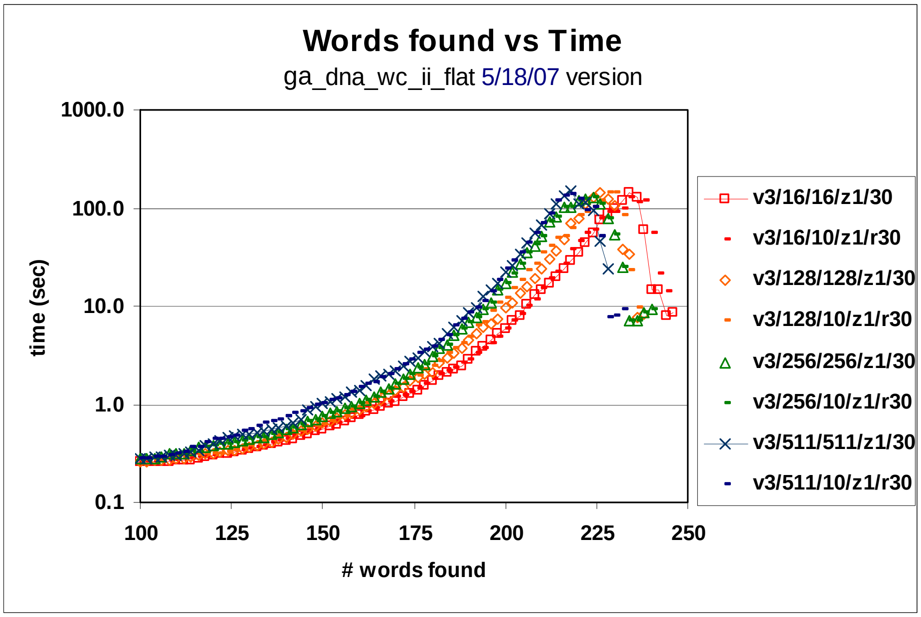


Figure 19. Performance effect of population size, version 4, time vs. # words.

Again we see that smaller populations find words faster. To help understand why, we modified this version to also report the generation on which words were found, as shown in Figure 20, which is for the same test as Figure 19. In both of these Figures, the results are averaged over 30 runs, (and so the average generation number can be a fraction). Also, there was a time limit of 5 minutes placed on the runs, so the point at the top right of these curves represent the average of only those runs that had not reached the 5 minute limit. The apparent reason why the curves dip is that runs that find a lot of words generally find them faster, too. While this is not important, we would like to understand why this happens. We had surmised that both of those observations about smaller populations were due to lower overhead in the sort operation, which would allow more random individuals to be checked by the mutation operator without being slowed down by the mating operator. Figure 20 can be interpreted to show that, indeed, this is the case.

The first thing to note in Figure 20 is that in terms of generations, all of the population sizes find words at about the same rate in the middle of the curves, e.g. word # 175 is found on about generation 9, regardless of the population size. Stated another way, runs with small population size find the same number of words at each generation that runs with larger population size do. It follows that if a generation takes longer to process (e.g. because the sort, checking after mating, and decloning operator execution times increase with population size), using large population simply penalizes run time. It is somewhat surprising that such a small population still finds words so efficiently. The reason might be the type of modified mutation operator we used, i.e. one that tries all 47 possible mutations. This means that every generation z x p * 47 mutations are checked, where z=% mutations, and p=population size. Even if z=1, p=16, this means 1 x 16 x 47 = 752 new mutations are checked each generation, whereas fewer than 16 new individuals are checked by the mating operator.

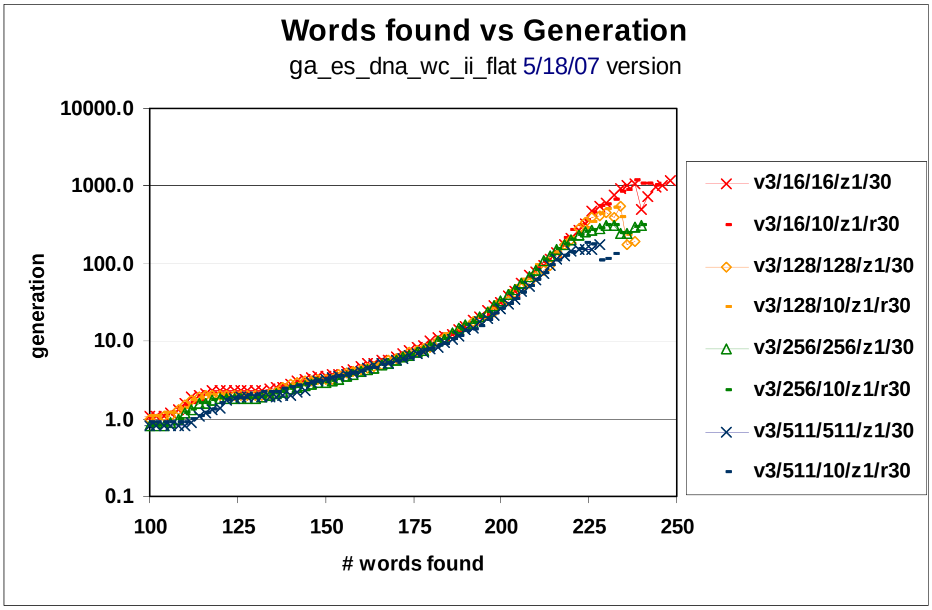


Figure 20. Performance effect of population size, version 4, generation vs. # words.

The second thing to note in Figure 20 is that the upper right tails of the curves show that runs with smaller population size find more words in the end than those for larger population sizes. Clearly, they also complete many more generations before they reach the time limit at the end of the test. No doubt the larger population runs would find just as many words, given more time to execute their longer generations.

The next experiment we did looked at the effect of selection pressure in the mating operator on performance. Figures 21 and 22 show the averaged results for 30 runs with population sizes of 32 and 256, respectively, and the number of keepers k varied in order to increase selection pressure. For these tests, mutation was minimized (1%), the population was decloned every 10 generations, and the termination time set to 30 seconds.

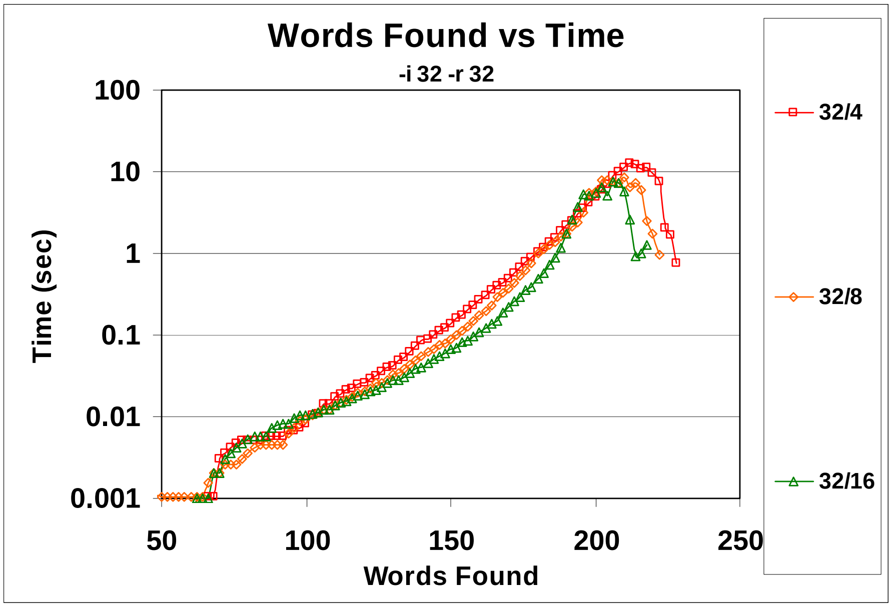


Figure 21.Population size 32, mating selection pressure varied, version 4.

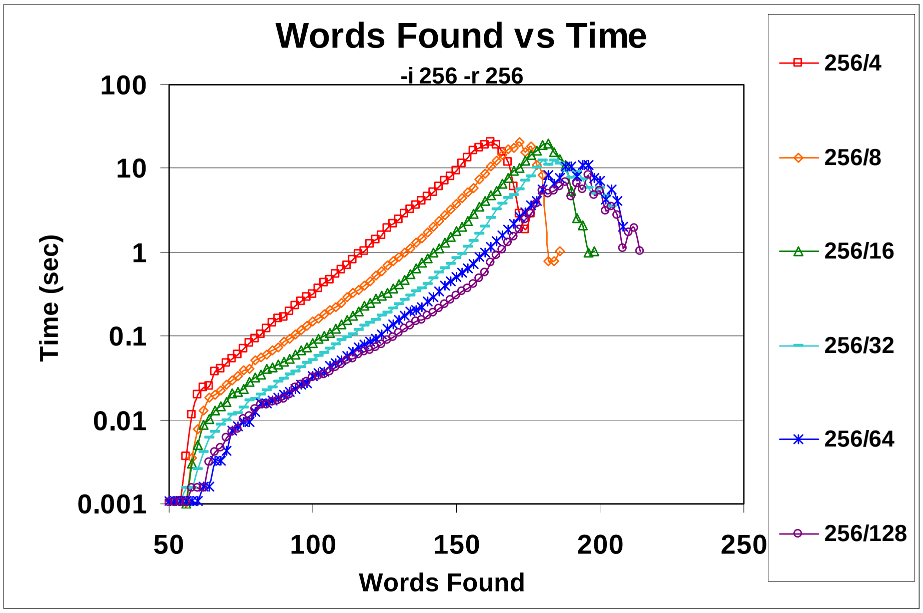


Figure 22. Population size 256, mating selection pressure varied, version 4.

The results in Figures 21 and 22 show that in the middle of the curves, increasing selection pressure (fewer # of keepers) actually degrades performance. All we can really say is that if there is a beneficial effect of increasing selection pressure in the mating operator, it is masked by a higher time cost of running the mating operator. However, at the right tails of the curves in Figure 21 (population size 32) we see that higher selection pressure found more words, but in Figure 22 (population size 256) lower selection pressure found more words. To understand these effects, next we looked at the execution times of the sort and selection/mating operators in as functions of population size and # keepers. We did this by inspecting waveforms produced by ModelSim simulations and constructing clock cycle models for these operators. We used these models to generate the curves shown in Figures 23-26.

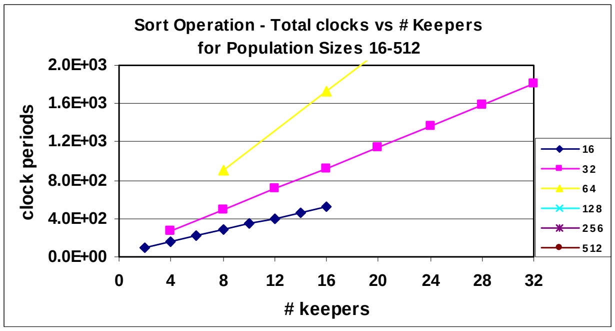


Figure 23. Sort clock cycles, population size 32. (curve 2nd from bottom).

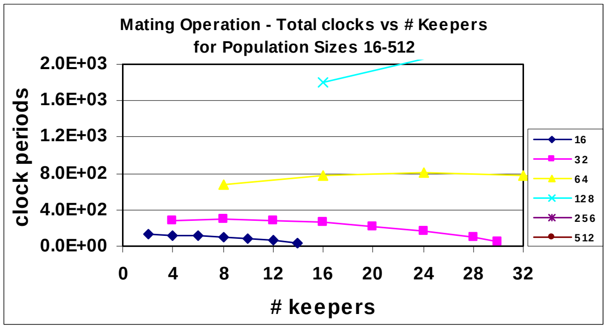


Figure 24. Selection/mating clock cycles, population size 32. (curve 2nd from bottom).

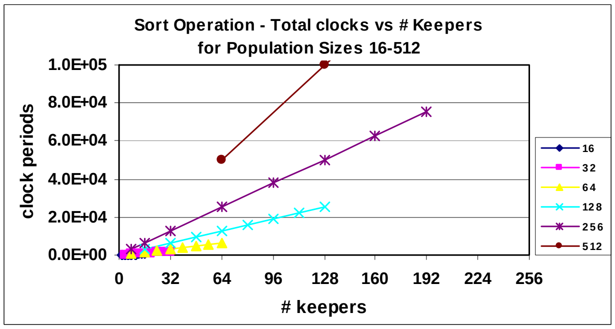


Figure 25. Sort clock cycles, population size 256. (curve 2nd from top).

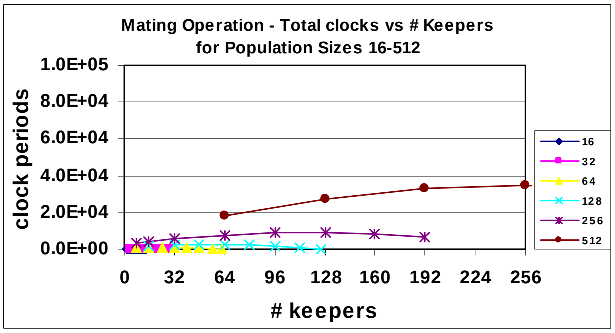


Figure 26. Selection/mating clock cycles, population size 256. (curve 2nd from top).

These curves show that the absolute time cost of sorting is much larger than that of selection/mating for all population sizes. They also show that sorting time increases linearly with the number of keepers, while selection/mating time is relatively constant vs. k. Finally, they show that the sensitivity of sorting time to increasing k is much larger for population size 256. Clearly, when these effects are taken together, we would expect a large beneficial effect on performance as the number of keepers is decreased. However, this is just the opposite of what we see in the middle of the curves of Figures 21 and 22, i.e. there is a degradation in performance as the number of keepers is decreased. This indicates that there must be another factor at work. At this writing we can only speculate that one possibility is that clones may be arising in the population more often for smaller k, which would also cause mutation trials to be on duplicate words. This would also cause slightly longer decloning times. Another way to think about this is that keeping a larger number of (presumably more diverse) individuals from generation to generation, rather than replacing them with a smaller number of individuals bred from a small number of parents, might effectively result in a wider search. This could be further studied by monitoring the # clones that appear, which could be done in future work.

One caveat is that certain assumptions had to be made when building the models, e.g. during the sort a number of passes are made through the population to pick up individuals with the best remaining fitness. We assumed that on average one individual would be picked up for each pass, meaning that NumKeepers/2 passes world occur. It is entirely possible that multiple individuals could have the same fitness and be picked up on the same pass, and that would tend to make the curves of Figures 23 and 25 flatter.

Finally, we note that the response to varying selection pressure may be very problem dependant. In the present case it appears that mating does not help performance, but mating may be very beneficial for other problem types, so it is good to have FPGA cores for doing it.

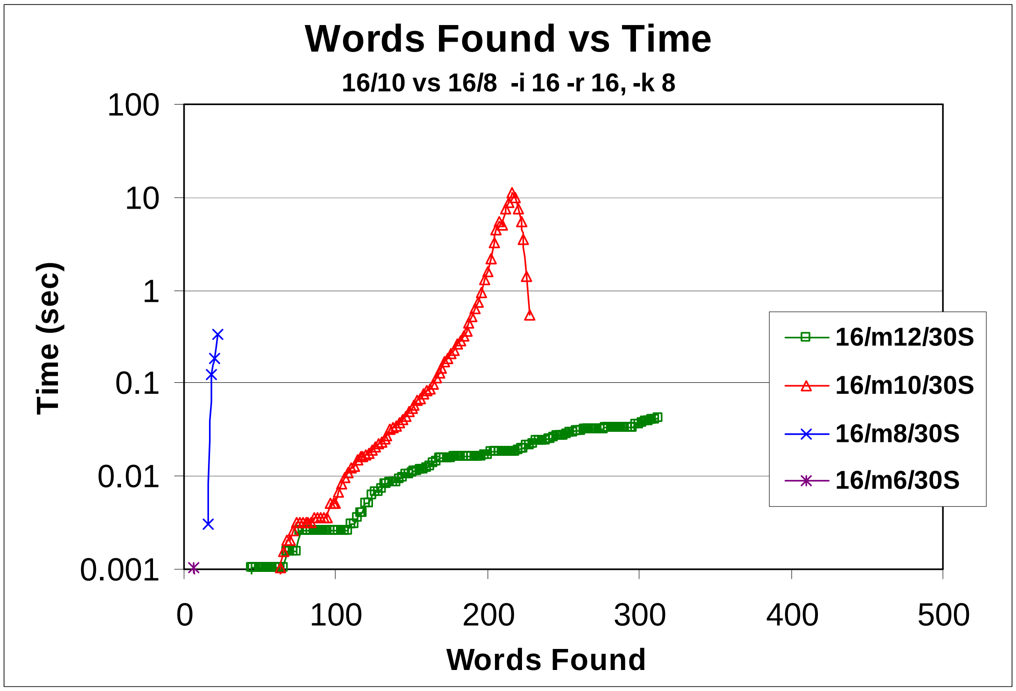


Figure 27. Edit distance varied, population size 16, # keepers 8, version 4.

Another experiment we did looked at the effect of the max_match parameter on the number of code words that can be found. The max_match parameter can be used to change the edit distance (which is code word length-max_match). The results Figure 27 show that more words are found as max_match increases. This is expected, because as max_match increases, the allowable edit distance decreases, meaning that the constraint on unintended cross hybridizations is less severe. More words can be added to the library when unintended cross hybridizations are allowed to be more likely that when they are constrained to be less likely.

Figure 28 shows a blow-up of the data in the lower left corner of Figure 27, plus another curve that is the average of 30 runs of 16/m8 with GA run for 20 minutes instead of 30 sec (which should have found more words if they could be found). We have seen that for 16/10 codes, on average GA finds over 98% of the words that can be found. We do not know if this is the case for 16/6 and 16/8 codes at this writing, because we have not done exhaustive search on the resulting codes. At any rate, this is some of the only data we have seen for these code length/distances.

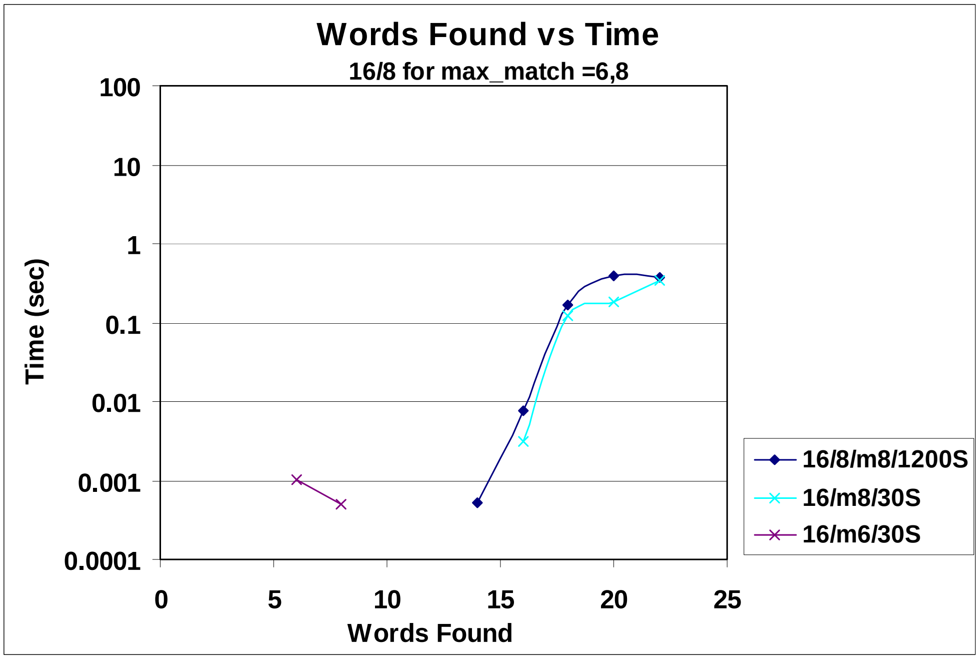


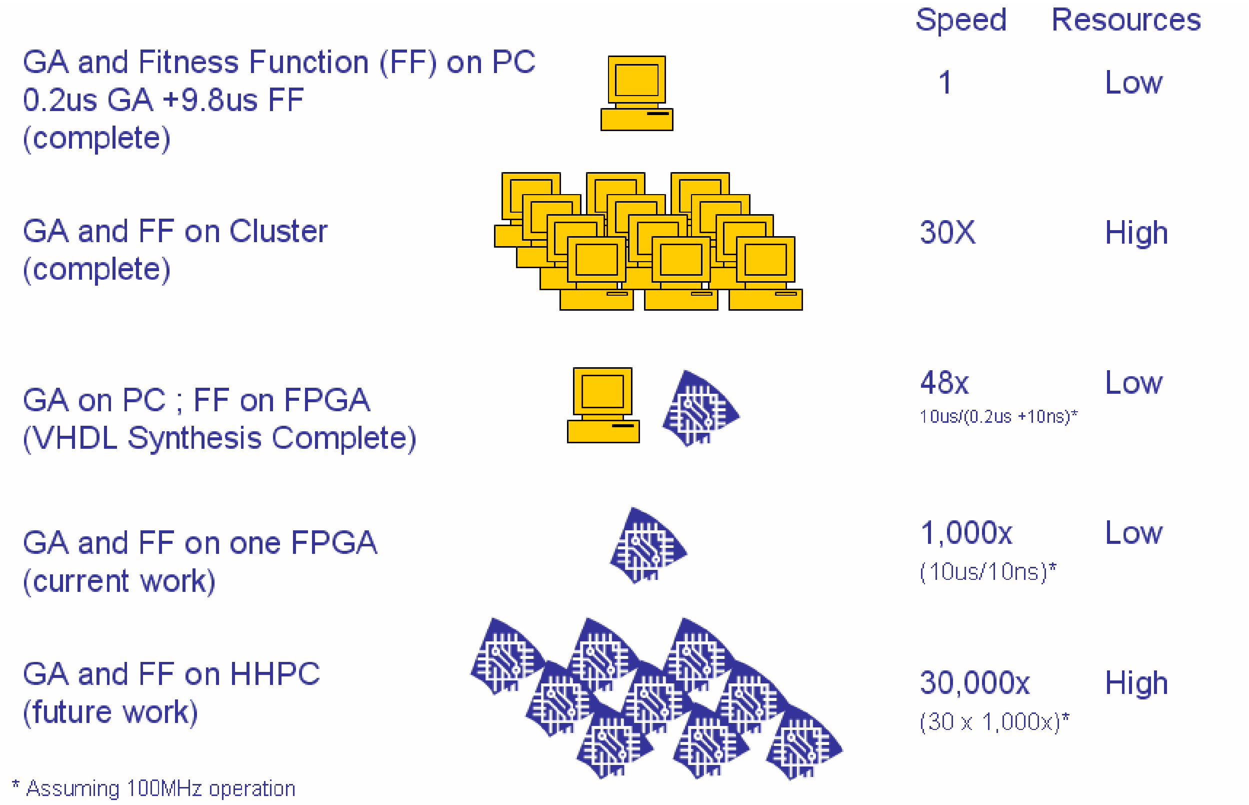
Figure 28. Max_match 6 and 8 codes (edit distance 10 and 8), version 4.

Finally, we note that some additional work is being done by our summer faculty to develop GA/DNA Code FPGA versions that incorporate additional features. That work is not covered in this report, but will be described elsewhere. One of these involves expanding the LLCS systolic array so that it computes on 32 mers instead of 16 mers. This is important because some biological applications of the LLCS metric involve computing on 25 mers, and even 60 mers. This preliminary work has successfully implemented and tested a 32 x 32 mer LLCS systolic array that fits on the WildStar FPGA, and operates at 45MHz. This version achieved about a 4,000X speed-up over software when building 32/20 codes, and about 100,000X speed-up composing 32/16 codes. These extreme speed-ups are due to the very long execution times of the software versions (which scale as n^2), and the fact that the execution time changes very little as the systolic array size is increased.

4.0 Conclusions and Future Work

We believe that this in-house project has accomplished its first goal, i.e. it demonstrated that the genetic algorithm, the workhorse evolutionary computing algorithm, can achieve results as good or better than classical methods, at least for the problem types we applied it to during this project (non-linear ODE parameterization, and DNA Code generation). We demonstrated this by building and testing prototype solution engines in spiral 1 in a variety of languages on a single workstation, in spiral 2 in C/MPI implementations on a cluster of workstations, and finally in spiral 3 on hybrid workstations equipped with special purpose boards hosting FPGAs. Part of this goal was to determine whether the evolutionary algorithm offered any advantages over classical methods, and in spiral 1 we did observe that for the non-linear model parameterization application the GA did converge when working with full, unreduced models, where a number of classical methods failed to converge unless reduced order models and step-wise parameter fitting methods were used. In the context of distributed, cluster based applications in spiral 2, again for the parameterization problem, we observed that the Island Model GA displayed approximately linear solution time speed-up vs. the number of nodes used, and also that it does not suffer from some of the limitations of alternative deterministic distributed global optimum seeking algorithms whose memory usage may grow exponentially if the solution space has many local optima. The sizes of the data structures required to run the GA are defined at time zero, and they do not change as the algorithm proceeds. Finally, in spiral 3 we were able to hand craft a full hardware implementations of a GA that was integrated with a DNA Code building application, and we were able to build and test several versions that explored variations of the GA operators and code building constraints. We compared our test results to those found in the literature, using both software GA and non-GA methods, and observe that the performance of our hardware GA/DNA Code building prototypes met or exceeded their speeds. We also observe that there are no other fully hardware implementations of any other algorithm for this application. Hardware implementations of the GA are in fact straight forward due to the simple, repetitive nature of the basic algorithm, and the regularity and parallelizability of its operators. For example, almost all memory accesses in our integrated algorithm/application are done to sequential, contiguous addresses, e.g. when processing the individuals in the GA population. We were also able to successfully exploit high speed data pipelining methods in 2D systolic arrays we designed for checking fitness with the LLCS and Gibbs energy metrics. That simply would not be possible for a number of other optimization algorithms that do one trial at a time, separated by (possibly time consuming) calculations to guide the choice of the next trial.

Figure 29 depicts a summary of the speed-up results obtained across the various platforms, including a couple of additional options that we did not prototype, but which we include because they might be important to look at in future work. At the beginning of this project, implementation on the third platform, GA on PC and FF (fitness function) in FPGA, would have suffered a severe a speed bottleneck associated with communication between the host PC and FPGA across the peripheral bus (33 or 66 MHz). However, at this writing at the end of the project, a new platform of this sort has appeared which has an FPGA chip in a socket on the CPU mother board which can use the high speed internal bus. Our organization is obtaining early version of this platform, and our GA/DNA code application may provide an early test case. If the communication rate for simultaneous input and output data streams between the CPU and FPGA can be maintained at about 100MHz, this platform may meet or exceed the performance of a 100MHz all FPGA solution. Finally, we note that time, workload, and urgency constraints conspired to prevent us from fully exploring the last platform shown, the cluster of FPGAs approach. Even so, we did develop a single chip prototype that demonstrated linear speed-up for a multi-deme GA, and another single chip prototype that simultaneously used two instances of the LLCS systolic array for exhaustive checking. Future work could address adapting these versions to run on a cluster of FPGAs, and we would expect both to exhibit linear speed-ups vs. the number of FPGA nodes. Finally, we find it extremely interesting to note that the single FPGA chip platform approach that we did study is far less expensive than the software on a cluster of workstations approach, yet it delivered far greater speed-ups. While this will not be true in general for any problem type, we expect it will hold true for those types of problems that involve fitness functions with simple mathematics (i.e. with integer and Boolean operands), especially where the function evaluation can be cast into a pipelined systolic array.



	Speed	Resources
GA and Fitness Function (FF) on PC 0.2us GA +9.8us FF (complete)	1	Low
GA and FF on Cluster (complete)	30X	High
GA on PC ; FF on FPGA (VHDL Synthesis Complete)	48x 10us/(0.2us +10ns)*	Low
GA and FF on one FPGA (current work)	1,000x (10us/10ns)*	Low
GA and FF on HHPC (future work)	30,000x (30 x 1,000x)*	High

* Assuming 100MHz operation

Figure 29. Speed-up and resources for the various platforms considered by this project.

Other goals of this project were to synergize with other ongoing in-house and summer faculty research topics, and the AFOSR and DARPA programs managed by RI, and to identify target problems in ongoing programs that might benefit from the development of new optimization tools for solving hard computational problems. We believe that we accomplished these goals by identifying and drawing our main application problems from both our ongoing involvements in the DARPA in the SIMBIOSYS and BIOCOMP programs, and our AFOSR sponsored projects working in the area of computing with bio-molecules, and nano-scaffold self-assembly for bio- and nano- electronics. Another a aspect of accomplishing this goal involved active interaction and collaborations with workers at Purdue, SUNY Geneseo, Wright State University, AFIT, SUNY Binghamton, and UNC Charlotte to obtain test case models, exchange ideas, compare results, and in general advocate the use of the methods investigated by this project. Finally, we synergized with our summer faculty programs by hosting three summer faculty and two summer students who contributed ideas and work to this project (see section 6.0 Acknowledgements).

Two other goals of this project were to raise the level of awareness of workers at RI about present day research and applications of evolutionary computing methods, and to mine the RI mission areas for additional candidate problems that could potentially benefit from this knowledge and the results of this effort. We believe that we accomplished these goals by helping to establish an EC Interest Group at RI that hosted a number of lectures by EC experts and practitioners, and by attending and publishing the results of this effort at numerous external Conferences and workshops. In addition to our main test case problems, we also identified and worked on at least three other problems to a lesser extent, including power management policy optimization in distributed sensor networks, low power bus coding for VLSI on-chip communication busses, and probe design for diagnostic micro-arrays.

Some of the most important technical accomplishments of this program involved the successful design and demonstration of working prototypes that were the first single chip FPGA solutions that integrated a hardware GA, and a separate hardware exhaustive search (ES) engine with a hardware version of a complex DNA Code generation application problem. The extreme speed-ups on the order of 1000X achieved by both of these designs enabled us to run experiments that yielded a first ever observation that GA alone can find about 99% of the DNA Code words that can be found. It is also very significant that the hardware ES makes experiments practical that simply would not be if done in software, for example, some of our routine tests that took 2 days with these hardware prototypes would have taken about 5.5 years in software. While it is true that a 1,000 node cluster could also have run these tests in 2 days in software (assuming that the performance scaling was perfectly linear), the FPGA platform we used costs significantly less ($1.8K plus a notebook computer), uses less power, and requires no facility.

We can think of several avenues to explore in future work. One would be to revisit the use of Higher Order Languages, and try to take more advantage of design tools that allow users to code in a C-like language, or even MatLab. We did succeed in producing a 16 x 16 LLCS fitness checking prototype early in spiral 3 using the Impulse C tool, which simulated correctly in the Impulse Co-Developer tool, and resulted in VHDL code. However, we were not able to obtain VHDL for the whole application, we believe because of limitation of the tool for extracting state machines from our very large application. Therefore, we were unable to synthesize and test code from this tool path, and we do not know how it would compare to our own hand crafted version in terms of resource utilization and speed. While success along these lines would probably increase the likelihood that researchers in various problem domain areas might pick up and use hardware acceleration methods, today this is not the general case, and hardware applications typically are done with help from a computer expert on the team.

Another possible area for future work would be to look at other GA operators and other EC algorithms, hopefully in the context of problems that are solved well, but that need speed-up.

Still another area for possible future would be to pursue even higher performance hardware implementations of the accelerators that we designed in this effort. For example, newer generation FPGA chips are now available that have in excess of 200,000 LUTs (e.g the largest Xilinx Virtex 5 chip). We note that our software GAs ran on 2.4GHz Pentium 4 workstations, and on our hardware GAs ran on Virtex-II and Virtex 4 FPGA chips at 30-120MHz. Future generations of both workstations and FPGAs will push both of these benchmarks upward. We also note that further improvements might be had by pursuing distributed GA implementations on multiple core workstations (2 and 4 core Intel and AMD processors are available at this writing), on the Sony playstation and IBM cell blade platforms, or on an Application Specific Integrated Circuit (ASIC) which would be expected to achieve perhaps 500MHz – 1.0 GHz clock speeds. We also note that at this writing there are no known commercially available EC chips that could act as a hardware accelerator for optimization problems, or even examples provided by FPGA vendors, so the cores we developed have the potential to help fill those gaps.

For this work we focused mainly on length 16, max_match 10 (16/10) DNA codes, and we generated only limited data with tests that composed 16/6, 16/8, and 16/12 codes. There might be academic and practical interest in using these prototypes to study codes generated with a variety of conditions, and codes with longer lengths (e.g. with our new prototype for 32 mers).

Finally, we note that a potentially high payoff spin-off application for the prototypes developed in this project has been identified. It is the acceleration of what is called the Probe Set Design problem that is encountered in the design of gene expression and gene identification diagnostic micro-arrays. This problem involves time consuming checking of similarity metrics between a set of short probe oligos (e.g. a set of thousands of 25 or 50-60 mers) against whole genomes. The goal is to identify a set of probes that can be placed in wells on a micro-array that when washed with an analyate with unknown genomic contents, will accurately identify the source organisms. The probe mers must be chosen carefully so that the patterns of bindings between the probes on the chip and the targets in the genomes can be interpreted unambiguously. There are various approaches that have been used to compose probe sets, and they involve computations of LLCS, Smith-Waterman similarity, and Gibbs energy free energy between the short mer probes and short mer segments drawn from target organism genomes. This problem is directly applicable to the task of designing and updating micro-arrays that can identify the presence of, say the top 20 biological threats. At present, we are exploring possibilities for applying our work in this area. To date we have attended a conference on the subject, written a C version of s simple form of the problem, and given a poster on our results [32]. We are also having discussions with workers at UNC Charlotte who have begun to implement a hardware accelerated Smith-Waterman algorithm [33].

5.0 Acknowledgements...

Kevin May, a Clarkson University senior, contributed greatly to the distributed GA ODE and DNA Code Word applications, and the VHDL fitness evaluator, while he was a student intern at RITC during the summers of 2003-2005, and winter breaks 2003 and 2004. Dr. Larry Merkle of Rose Hulman Institute of Technology contributed ideas for the software distributed GA and hardware GA Core during a Visiting Summer Faculty Research assignment at RI in the summer of 2004. Clare Thiem/RITC provided interface to the DARPA BIOCOMP and SIMBIOSYS programs. The DARPA SIMBIOSYS program manager (Dr. Anantha Krisnan), and the DARPA BIOCOMP program manager (Dr. Sri Kumar) provided partial support for this work through agent fees, with help from the local RI program contact for the BIOCOMP program (Dr. Robert Kaminski/RIGA). Dr. Thomas Renz, RITC lead for the BioMolecular Computing Program, provided much inspiration and support. Ann Rundell, Purdue University, provided MatLab codes for the Ag-Ab binding model during spirals 1 and 2. Dr. John Gallagher, Wright State University, and Prof. Gary Lamont, Air Force Institute of Technology, provided references and valuable discussions about compact/mini-pop GAs, and multi-objective optimization using GA, respectively. Dr. Tony Macula, SUNY Geneseo, and Morgan Bishop, JEANSEE Corp., Geneseo, NY, contributed the Markov DNA Code Word generation results. David Pelliren, Impulse Accelerated Technologies, provided assistance in evaluating Co_Developer software. Dr. Qinru Qiu, Dept. ECE, State University of NY at Binghamton, contributed the work on the Sensor Network Energy Management Problem. Dr. Qinru Qiu, Dr. Qing Wu, and Prakash Mukre at SUNY Binghamton contributed greatly to the implementation of the FPGA prototypes. Andrew Flack, an undergraduate student at the University of Rochester, contributed to the study of the application of the results of this project to the probe design problem.

References

[1] D.J. Burns, "Hybrid Architectures for Evolutionary Computing Algorithms", In-house Interim Report, AFRL-IF-RS-TR-2006-14, DTIC Accession No: ADA444730, Jan. 2006.
http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA444730

[2] A.E. Rundel, H. HogenEsch, T.J. Webstster, "Optimizing the Immuno-Surface Characteristics for Bio-Sensors and Filters Through Modeling and Experiments", Final Technical Report, DTIC Accession # ADA436117, June 2005.
http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA436117

[3] J.J Tyson, B. Novak, K. Chen, J.C. Sible, F.R. Cross, L.T. Watson, C.A. Shaffer, "Eukaryotic Cell Cycle as a Test Case for Modeling Cellular Regulation in a Collaborative Problem-Solving Environment", Final Technical Report, DTIC Accession # ADA464845, Mar. 2007.
http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA464845

[4] P. Mendes, "GEPASI: A Software Package for Modeling Dynamics, Steady States and Control of Biochemical and Other Systems", Comp. Applic. Biosci., 9, 1993, pp. 563-571.
http://bioinformatics.oxfordjournals.org/cgi/content/abstract/9/5/563 or http://www.gepasi.org/

[5] M. Rodriguez-Fernandez, J.A. Egea, and J.R. Banga, "Novel metaheuristic for parameter estimation in nonlinear dynamic biological systems", BMC Bioinformatics, 2006; 7: 483, published online 2006 November 2.
http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1654195

[6] L. Brus, "Recursive Black-box Identification of Nonlinear State-space ODE Models", IT Licentiate Thesis, 2006-001, , Department of Information Technology, UPPSALA UNIVERSITY, Sweden, Jan. 2006.
http://www.it.uu.se/research/publications/lic/2006-001/2006-001.pdf

[7] J.W. Zwolak, J.J. Tyson and L.T. Watson, Globally Optimized Parameters for a Model of Mitotic Control in Frog Eggs", IEE Proc.-Syst. Biol., Vol. 152, No. 2, June 2005, pp. 81-92. http://mpf.biol.vt.edu/people/jzwolak/papers/iee05.pdf

[8] J.C. Meza, R.S. Judson, T.R. Faulkner, and A.M. Treasurywala, "A comparison of a direct search method and a genetic algorithm for conformational searching", J. Comput. Chem., 1996, 17, (9), pp. 1142–1151. http://crd.lbl.gov/~meza/papers/gavspds_jcc.pdf

[9] T.T.Hanp LALONG, Q.Tuan PHAM, "A Comparison of the Performance of Classical Methods and Genetic Algorithms for Optimization Problems Involving Numerical Models", pp. 2019-2025. http://ieeexplore.ieee.org/iel5/9096/28878/01299921.pdf

[10] http://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization

[11] D. J. Burns, K.T. May, "On Parameterizing Models of Antigen-Antibody Binding Dynamics on Surfaces – a Progressive Genetic Algorithm Approach and the Need for Speed", Genetic and Evolutionary Computing Conference - 2004, Seattle, WA, June 2004, in Lecture Notes in Computer Science, Springer Berlin/Heidlberg, Vol. 3102/2004, pp. 497-498.
http://www.springerlink.com/content/ucg2e2ybt6t33v3g/fulltext.pdf see also
http://www.rose-hulman.edu/~merkle/Professional/Publications%20and%20Presentations/Others/2004_08%20VFRP.pdf

[12] D.J. Burns, K.T. May, M. Bishop, "DNA Code Word Library Generation Using a Parallel Genetic Algorithm", Foundations of Nanoscience – Self Assembled Architectures and Devices (FNANO 2005), Snowbird, UT, Apr. 2005, pp 128-129. (available from http://sciencetechnica.com/)

[13] DJ.. Burns, "Hybrid Architectures for Evolutionary Computing Algorithms", EC In Practice session presentation, Genetic and Evolutionary Computation Conference, GECCO-2005, Washington, DC, June 2005.

[14] D.J. Burns, K. May, M. Bishop, "DNA Codeword Library Design Using a Parallel Genetic Algorithm", Workshop on Military and Security Applications of Evolutionary Computing, Genetic and Evolutionary Computation Conference, GECCO-2005, Washington, DC, June 2005.

[15] K.T May, "DNA Codeword Library Design Using A Parallel Genetic Algorithm with FPGA Hardware Fitness Function Co-Processor", GECCO 2005 Undergraduate Student Workshop Washington, DC25 June 2005.

[16] Qinru Qiu, Qing Wu, D. Burns, D. Holzhauer, "Distributed Genetic Algorithm for Energy-Efficient Resource Management in Sensor Networks", GECCO 2006, open poster session, Seattle, WA, July, 2006

[17] Qinru Qiu, Qing Wu, D. Burns, D. Holtzhauer, "Lifetime Aware Resource Management for Sensor Networks Using a Distributed Genetic Algorithm", IEEE/ACM Intl. Symposium on Low Power Electronics and Design (ISLPD-2006) Tegernsee, Germany Oct, 2006. (See Appendix A).

[18] D. Burns, K. May, T. Renz, V. Ross, "Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis", 2005 Science and Technology Algorithm Workshop (STAR 2005), Dayton, OH, 30 Aug – 1 Sep 2005.

[19] D. Burns, K. May, T. Renz, V. Ross, "Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis", 8th

Military and Aerospace Programmable Logic Devices (MAPLD) International Conference (2005) Washington, DC Sep 2005. (See Appendix B).

[20] http://www.xilinx.com/ise/verification/mxe_details.html

[21] http://www.synplicity.com/products/synplifypro/

[22] http://www.xilinx.com/ise/logic_design_prod/webpack.htm .

[23] D. Burns, M. Bishop, "FPGA Implementation of Systolic Array Levenshtein Matrix Distance Calculator for Reverse complement DNA Code Design ", Proceedings of the Workshop on Military and Security Applications of Evolutionary Computing, ASM Genetic and Evolutionary Computing Conference (GECCO 2006), Seattle. WA, July 2006, Distributed on CD-Rom at GECCO 2006. (See Appendix C).

[24] D. Burns, Qinru Qiu, Qing Wu, "FPGA Implementation of Systolic Array Levenshtein Matrix Distance Calculator for Reverse complement DNA Code Design", Poster Session, 2006 Military and Aerospace Applications of Programmable Logic Devices International Conference (2006 MAPLD), Washington, DC, Sep. 2006. (See Appendix D).

[25] Q. Qiu, D. Burns, Q. Wu, P. Mukre, "Hybrid Architecture for Accelerating DNA Codeword Searching", Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB 2007), IEEE Symposium Series on Computational Intelligence (SSCI-2007), Honolulu, HI, Apr. 2007. (See Appendix E).

[26] IEEE Journal of Computational Intelligence, (invited, in preparation).

[27] Q. Qiu, D. Burns, P. Mukre, Q. Wu, "Hardware Acceleration of Multi-deme Genetic Algorithm for the Application of DNA Codeword Searching", to appear in the Proceedings of the 2007 Genetic and Evolutionary Computing Conference (GECCO 2007), London, UK, July 2007. (See Appendix F).

[28] Q. Qiu, P. Mukre, M. Bishop, D. Burns, Q. Wu, "Hardware Acceleration for Thermodynamically Constrained DNA Codeword Searching Hardware Acceleration of Multi-deme Genetic Algorithm for the Application of DNA Codeword Searching", to appear in the Proceedings of The 13th Intl. Meeting on DNA Computing (DNA 13), Memphis, TN, June 2007. (See Appendix G).

[29] Springer Lecture Notes in Computer Science (invited, in press)

[30] GECCO 2008 (in preparation).

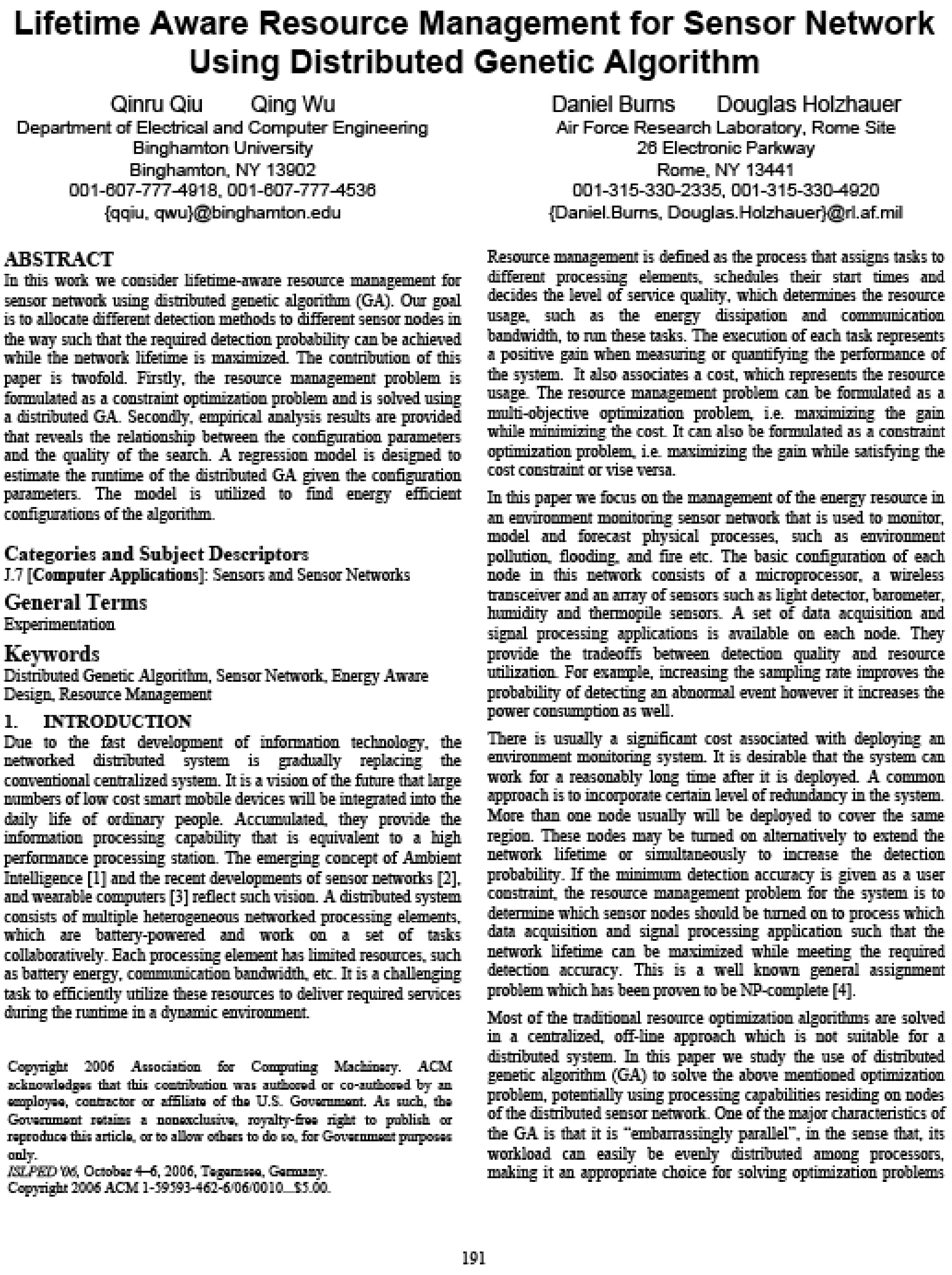
[31] Bishop, M. , Macula, A. , Pogozelski, W. , and Rykov, V. , "DNA Codeword Library Design", Foundations of Nanoscience – Self Assembled Architectures and Devices (FNANO 2005), Snowbird, UT, Apr. 2005, p 49. (available from http://sciencetechnica.com/)

[32] D. Burns, Q. Qiu, Q. Wu, and A. Flack, "On the Hardware Acceleration of the Probe Design Problem", 6th Annual System Integration in Biodefense Conference (poster paper), Washington, DC, Aug. 2007. (See Appendix H).

[33] R. Karanam, A. Ravindran, A. Mukherjee, C. Gibas, and A. Wilkison, "Using FPGA-based Hybrid Computers for Bioinformatics Applications", XCell Journal, Issue 58, Third Quarter, 2006. http://www.xilinx.com/publications/xcellonline/xcell_58/xc_pdf/p080-083_58-dna.pdf

Appendices

Appendix A: ISLPED 2006 paper. [Reference 17]. (Click to view Appendix_A.pdf).



Lifetime Aware Resource Management for Sensor Network Using Distributed Genetic Algorithm

Qinru Qiu Qing Wu
Department of Electrical and Computer Engineering
Binghamton University
Binghamton, NY 13902
001-607-777-4918, 001-607-777-4536
{qqiu, qwu}@binghamton.edu

Daniel Burns Douglas Holzhauer
Air Force Research Laboratory, Rome Site
26 Electronic Parkway
Rome, NY 13441
001-315-330-2335, 001-315-330-4920
{Daniel.Burns, Douglas.Holzhauer}@rl.af.mil

ABSTRACT

In this work we consider lifetime-aware resource management for sensor network using distributed genetic algorithm (GA). Our goal is to allocate different detection methods to different sensor nodes in the way such that the required detection probability can be achieved while the network lifetime is maximized. The contribution of this paper is twofold. Firstly, the resource management problem is formulated as a constraint optimization problem and is solved using a distributed GA. Secondly, empirical analysis results are provided that reveals the relationship between the configuration parameters and the quality of the search. A regression model is designed to estimate the runtime of the distributed GA given the configuration parameters. The model is utilized to find energy efficient configurations of the algorithm.

Categories and Subject Descriptors

J.7 [**Computer Applications**]: Sensors and Sensor Networks

General Terms

Experimentation

Keywords

Distributed Genetic Algorithm, Sensor Network, Energy Aware Design, Resource Management

1. INTRODUCTION

Due to the fast development of information technology, the networked distributed system is gradually replacing the conventional centralized system. It is a vision of the future that large numbers of low cost smart mobile devices will be integrated into the daily life of ordinary people. Accumulated, they provide the information processing capability that is equivalent to a high performance processing station. The emerging concept of Ambient Intelligence [1] and the recent developments of sensor networks [2], and wearable computers [3] reflect such vision. A distributed system consists of multiple heterogeneous networked processing elements, which are battery-powered and work on a set of tasks collaboratively. Each processing element has limited resources, such as battery energy, communication bandwidth, etc. It is a challenging task to efficiently utilize these resources to deliver required services during the runtime in a dynamic environment.

Resource management is defined as the process that assigns tasks to different processing elements, schedules their start times and decides the level of service quality, which determines the resource usage, such as the energy dissipation and communication bandwidth, to run these tasks. The execution of each task represents a positive gain when measuring or quantifying the performance of the system. It also associates a cost, which represents the resource usage. The resource management problem can be formulated as a multi-objective optimization problem, i.e. maximizing the gain while minimizing the cost. It can also be formulated as a constraint optimization problem, i.e. maximizing the gain while satisfying the cost constraint or vise versa.

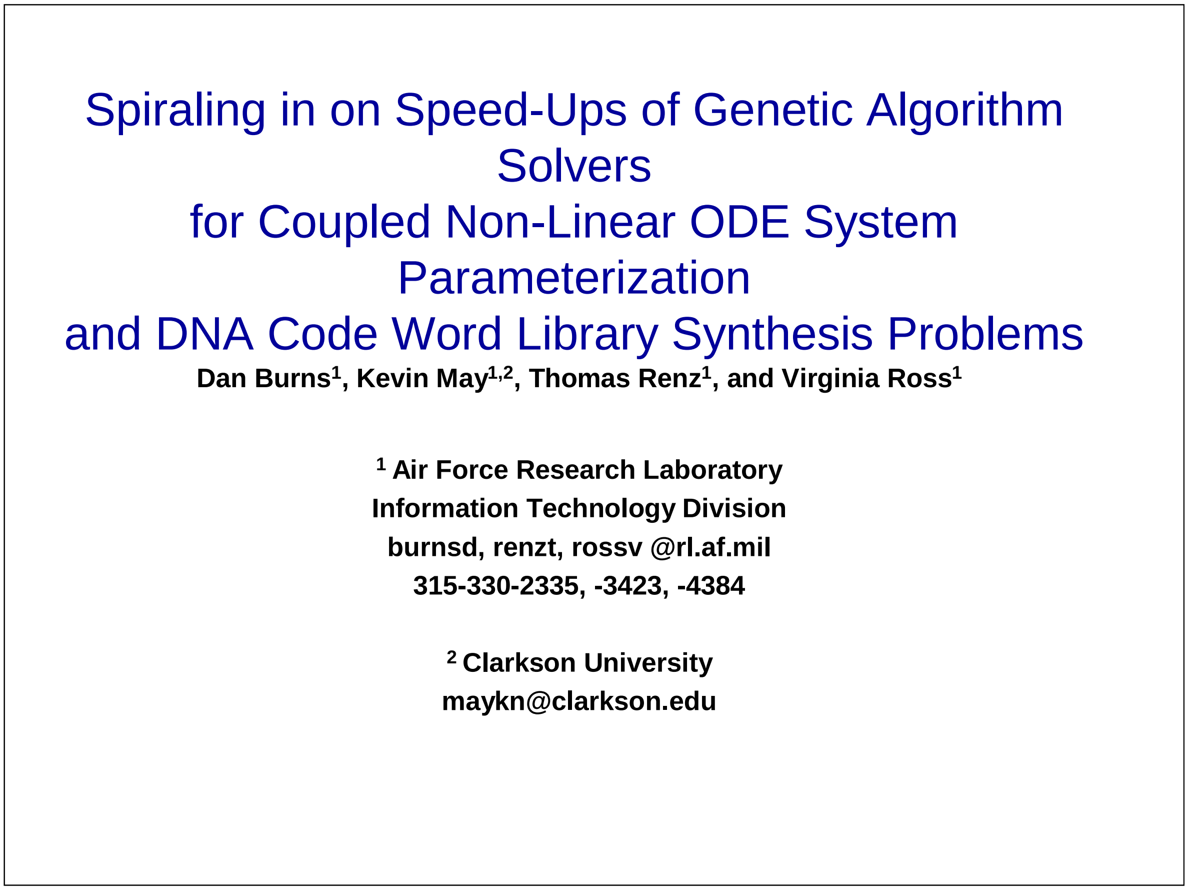
In this paper we focus on the management of the energy resource in an environment monitoring sensor network that is used to monitor, model and forecast physical processes, such as environment pollution, flooding, and fire etc. The basic configuration of each node in this network consists of a microprocessor, a wireless transceiver and an array of sensors such as light detector, barometer, humidity and thermopile sensors. A set of data acquisition and signal processing applications is available on each node. They provide the tradeoffs between detection quality and resource utilization. For example, increasing the sampling rate improves the probability of detecting an abnormal event however it increases the power consumption as well.

There is usually a significant cost associated with deploying an environment monitoring system. It is desirable that the system can work for a reasonably long time after it is deployed. A common approach is to incorporate certain level of redundancy in the system. More than one node usually will be deployed to cover the same region. These nodes may be turned on alternatively to extend the network lifetime or simultaneously to increase the detection probability. If the minimum detection accuracy is given as a user constraint, the resource management problem for the system is to determine which sensor nodes should be turned on to process which data acquisition and signal processing application such that the network lifetime can be maximized while meeting the required detection accuracy. This is a well known general assignment problem which has been proven to be NP-complete [4].

Most of the traditional resource optimization algorithms are solved in a centralized, off-line approach which is not suitable for a distributed system. In this paper we study the use of distributed genetic algorithm (GA) to solve the above mentioned optimization problem, potentially using processing capabilities residing on nodes of the distributed sensor network. One of the major characteristics of the GA is that it is "embarrassingly parallel", in the sense that, its workload can easily be evenly distributed among processors, making it an appropriate choice for solving optimization problems

ISLPED '06, October 4–6, 2006, Tegernsee, Germany.
Copyright 2006 ACM 1-59593-462-6/06/0010...$5.00.

Appendix B: MAPLD 2005 paper [Reference 19]. (Click to view Appendix_B.pdf).



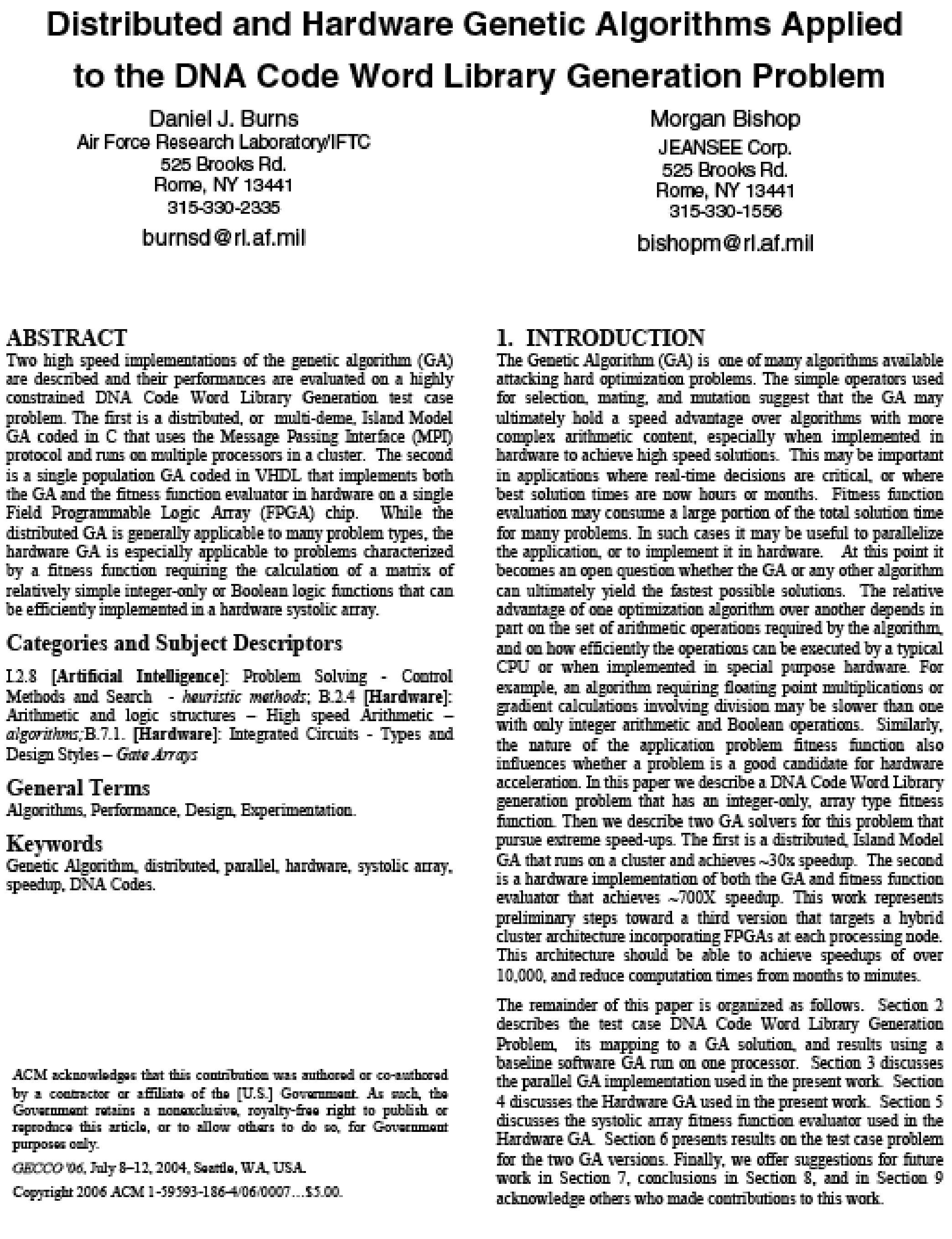
Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis Problems

Dan Burns[1], Kevin May[1,2], Thomas Renz[1], and Virginia Ross[1]

[1] Air Force Research Laboratory
Information Technology Division
burnsd, renzt, rossv @rl.af.mil
315-330-2335, -3423, -4384

[2] Clarkson University
maykn@clarkson.edu

Appendix C. GECCO 2006 paper. [Reference 23]. (Click to view Appendix_C.pdf).



Distributed and Hardware Genetic Algorithms Applied to the DNA Code Word Library Generation Problem

Daniel J. Burns
Air Force Research Laboratory/IFTC
525 Brooks Rd.
Rome, NY 13441
315-330-2335
burnsd@rl.af.mil

Morgan Bishop
JEANSEE Corp.
525 Brooks Rd.
Rome, NY 13441
315-330-1556
bishopm@rl.af.mil

ABSTRACT

Two high speed implementations of the genetic algorithm (GA) are described and their performances are evaluated on a highly constrained DNA Code Word Library Generation test case problem. The first is a distributed, or multi-deme, Island Model GA coded in C that uses the Message Passing Interface (MPI) protocol and runs on multiple processors in a cluster. The second is a single population GA coded in VHDL that implements both the GA and the fitness function evaluator in hardware on a single Field Programmable Logic Array (FPGA) chip. While the distributed GA is generally applicable to many problem types, the hardware GA is especially applicable to problems characterized by a fitness function requiring the calculation of a matrix of relatively simple integer-only or Boolean logic functions that can be efficiently implemented in a hardware systolic array.

Categories and Subject Descriptors

I.2.8 **[Artificial Intelligence]**: Problem Solving - Control Methods and Search - *heuristic methods*; B.2.4 **[Hardware]**: Arithmetic and logic structures – High speed Arithmetic – *algorithms;*B.7.1. **[Hardware]**: Integrated Circuits - Types and Design Styles – *Gate Arrays*

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Genetic Algorithm, distributed, parallel, hardware, systolic array, speedup, DNA Codes.

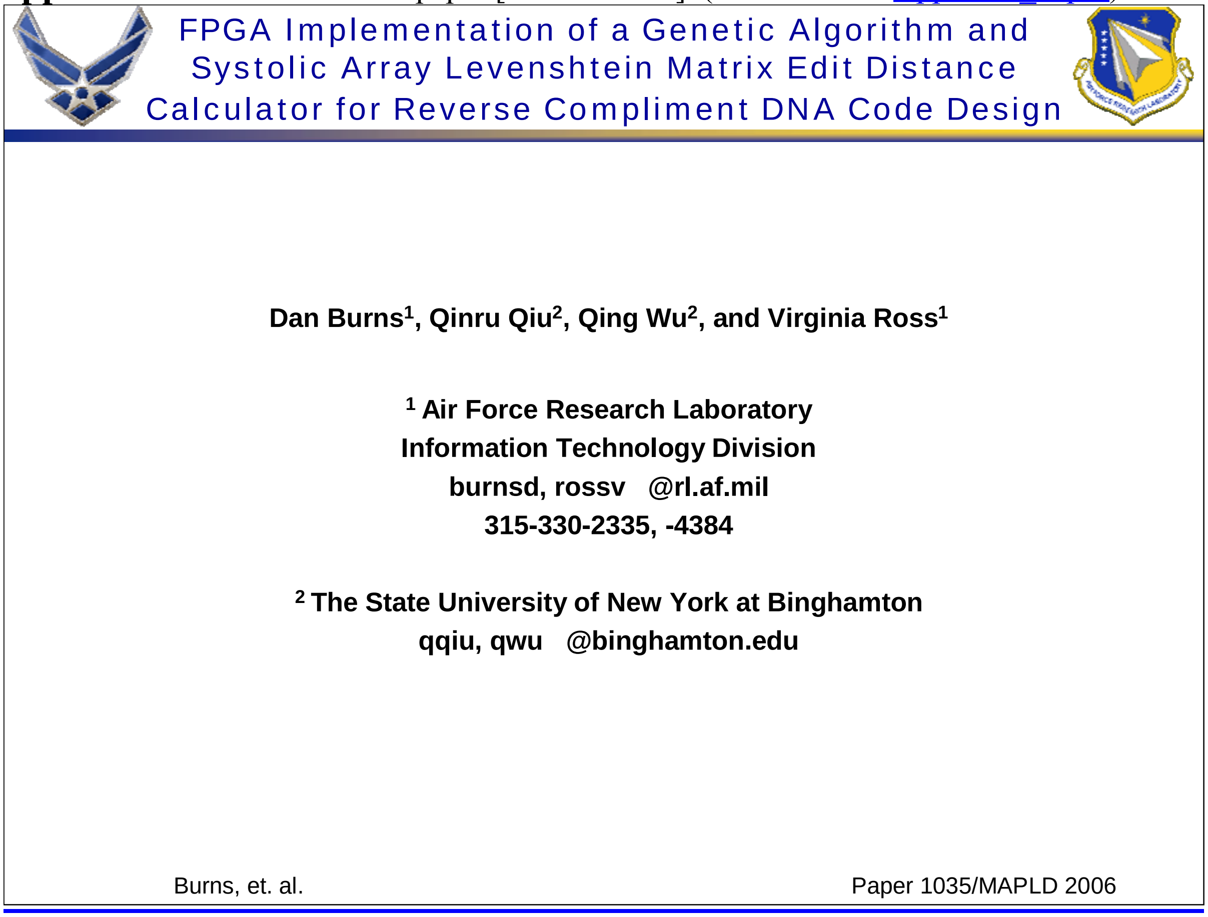
GECCO'06, July 8–12, 2004, Seattle, WA, USA.

1. INTRODUCTION

The Genetic Algorithm (GA) is one of many algorithms available attacking hard optimization problems. The simple operators used for selection, mating, and mutation suggest that the GA may ultimately hold a speed advantage over algorithms with more complex arithmetic content, especially when implemented in hardware to achieve high speed solutions. This may be important in applications where real-time decisions are critical, or where best solution times are now hours or months. Fitness function evaluation may consume a large portion of the total solution time for many problems. In such cases it may be useful to parallelize the application, or to implement it in hardware. At this point it becomes an open question whether the GA or any other algorithm can ultimately yield the fastest possible solutions. The relative advantage of one optimization algorithm over another depends in part on the set of arithmetic operations required by the algorithm, and on how efficiently the operations can be executed by a typical CPU or when implemented in special purpose hardware. For example, an algorithm requiring floating point multiplications or gradient calculations involving division may be slower than one with only integer arithmetic and Boolean operations. Similarly, the nature of the application problem fitness function also influences whether a problem is a good candidate for hardware acceleration. In this paper we describe a DNA Code Word Library generation problem that has an integer-only, array type fitness function. Then we describe two GA solvers for this problem that pursue extreme speed-ups. The first is a distributed, Island Model GA that runs on a cluster and achieves ~30x speedup. The second is a hardware implementation of both the GA and fitness function evaluator that achieves ~700X speedup. This work represents preliminary steps toward a third version that targets a hybrid cluster architecture incorporating FPGAs at each processing node. This architecture should be able to achieve speedups of over 10,000, and reduce computation times from months to minutes.

The remainder of this paper is organized as follows. Section 2 describes the test case DNA Code Word Library Generation Problem, its mapping to a GA solution, and results using a baseline software GA run on one processor. Section 3 discusses the parallel GA implementation used in the present work. Section 4 discusses the Hardware GA used in the present work. Section 5 discusses the systolic array fitness function evaluator used in the Hardware GA. Section 6 presents results on the test case problem for the two GA versions. Finally, we offer suggestions for future work in Section 7, conclusions in Section 8, and in Section 9 acknowledge others who made contributions to this work.

Appendix D. MAPLD 2006 paper [Reference 24]. (Click to view Appendix_D.pdf).



FPGA Implementation of a Genetic Algorithm and Systolic Array Levenshtein Matrix Edit Distance Calculator for Reverse Compliment DNA Code Design

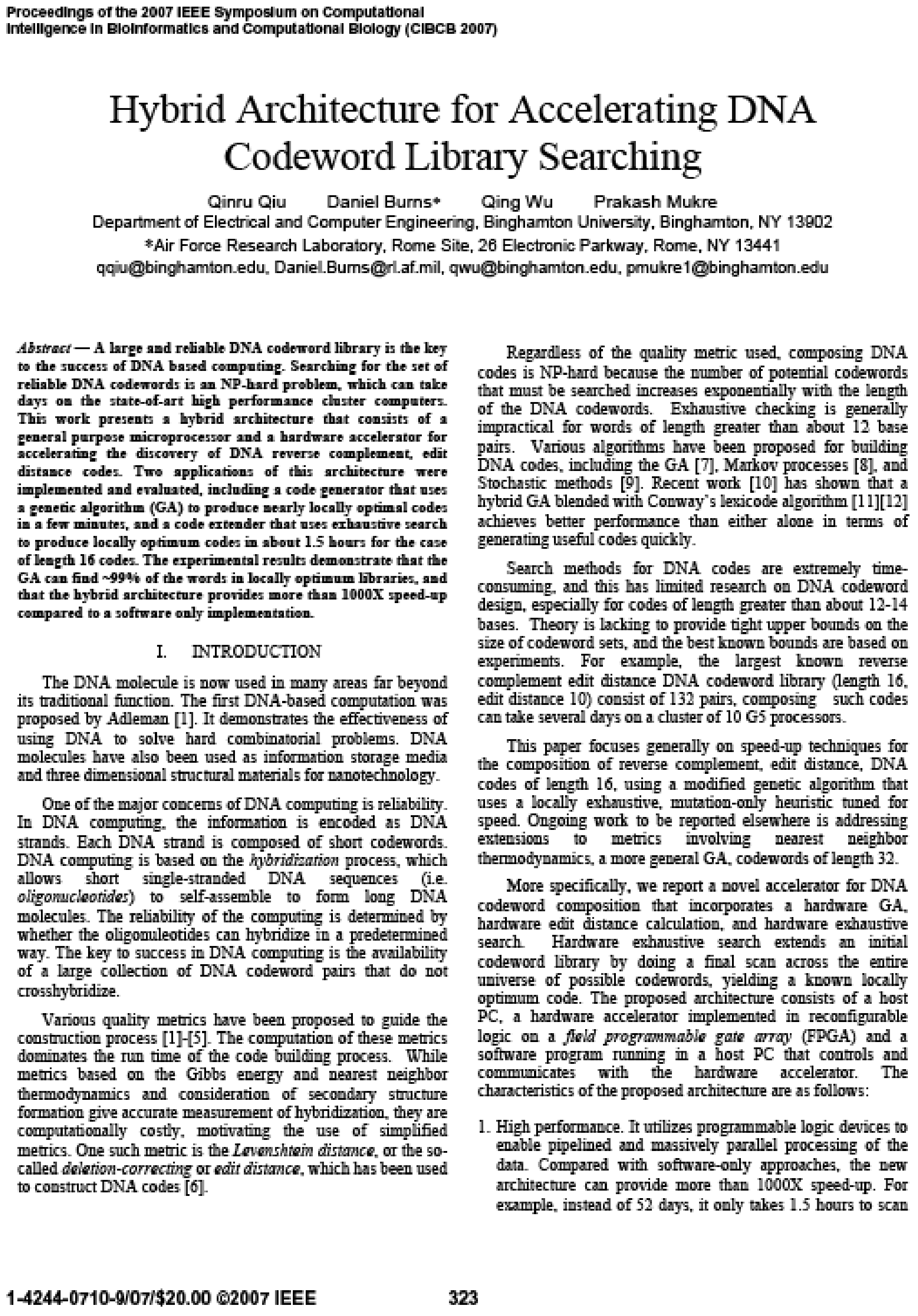
Dan Burns[1], Qinru Qiu[2], Qing Wu[2], and Virginia Ross[1]

[1] Air Force Research Laboratory
Information Technology Division
burnsd, rossv @rl.af.mil
315-330-2335, -4384

[2] The State University of New York at Binghamton
qqiu, qwu @binghamton.edu

Burns, et. al. Paper 1035/MAPLD 2006

Appendix E. SCCI 2007 paper [Reference 25]. (Click to view Appendix_E.pdf).



Proceedings of the 2007 IEEE Symposium on Computational
Intelligence in Bioinformatics and Computational Biology (CIBCB 2007)

Hybrid Architecture for Accelerating DNA Codeword Library Searching

Qinru Qiu Daniel Burns* Qing Wu Prakash Mukre
Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902
*Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441
qqiu@binghamton.edu, Daniel.Burns@rl.af.mil, qwu@binghamton.edu, pmukre1@binghamton.edu

Abstract **— A large and reliable DNA codeword library is the key to the success of DNA based computing. Searching for the set of reliable DNA codewords is an NP-hard problem, which can take days on the state-of-art high performance cluster computers. This work presents a hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the discovery of DNA reverse complement, edit distance codes. Two applications of this architecture were implemented and evaluated, including a code generator that uses a genetic algorithm (GA) to produce nearly locally optimal codes in a few minutes, and a code extender that uses exhaustive search to produce locally optimum codes in about 1.5 hours for the case of length 16 codes. The experimental results demonstrate that the GA can find ~99% of the words in locally optimum libraries, and that the hybrid architecture provides more than 1000X speed-up compared to a software only implementation.**

I. INTRODUCTION

The DNA molecule is now used in many areas far beyond its traditional function. The first DNA-based computation was proposed by Adleman [1]. It demonstrates the effectiveness of using DNA to solve hard combinatorial problems. DNA molecules have also been used as information storage media and three dimensional structural materials for nanotechnology.

One of the major concerns of DNA computing is reliability. In DNA computing, the information is encoded as DNA strands. Each DNA strand is composed of short codewords. DNA computing is based on the *hybridization* process, which allows short single-stranded DNA sequences (i.e. *oligonucleotides*) to self-assemble to form long DNA molecules. The reliability of the computing is determined by whether the oligonuleotides can hybridize in a predetermined way. The key to success in DNA computing is the availability of a large collection of DNA codeword pairs that do not crosshybridize.

Various quality metrics have been proposed to guide the construction process [1]-[5]. The computation of these metrics dominates the run time of the code building process. While metrics based on the Gibbs energy and nearest neighbor thermodynamics and consideration of secondary structure formation give accurate measurement of hybridization, they are computationally costly, motivating the use of simplified metrics. One such metric is the *Levenshtein distance*, or the so-called *deletion-correcting* or *edit distance*, which has been used to construct DNA codes [6].

Regardless of the quality metric used, composing DNA codes is NP-hard because the number of potential codewords that must be searched increases exponentially with the length of the DNA codewords. Exhaustive checking is generally impractical for words of length greater than about 12 base pairs. Various algorithms have been proposed for building DNA codes, including the GA [7], Markov processes [8], and Stochastic methods [9]. Recent work [10] has shown that a hybrid GA blended with Conway's lexicode algorithm [11][12] achieves better performance than either alone in terms of generating useful codes quickly.

Search methods for DNA codes are extremely time-consuming, and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. Theory is lacking to provide tight upper bounds on the size of codeword sets, and the best known bounds are based on experiments. For example, the largest known reverse complement edit distance DNA codeword library (length 16, edit distance 10) consist of 132 pairs, composing such codes can take several days on a cluster of 10 G5 processors.

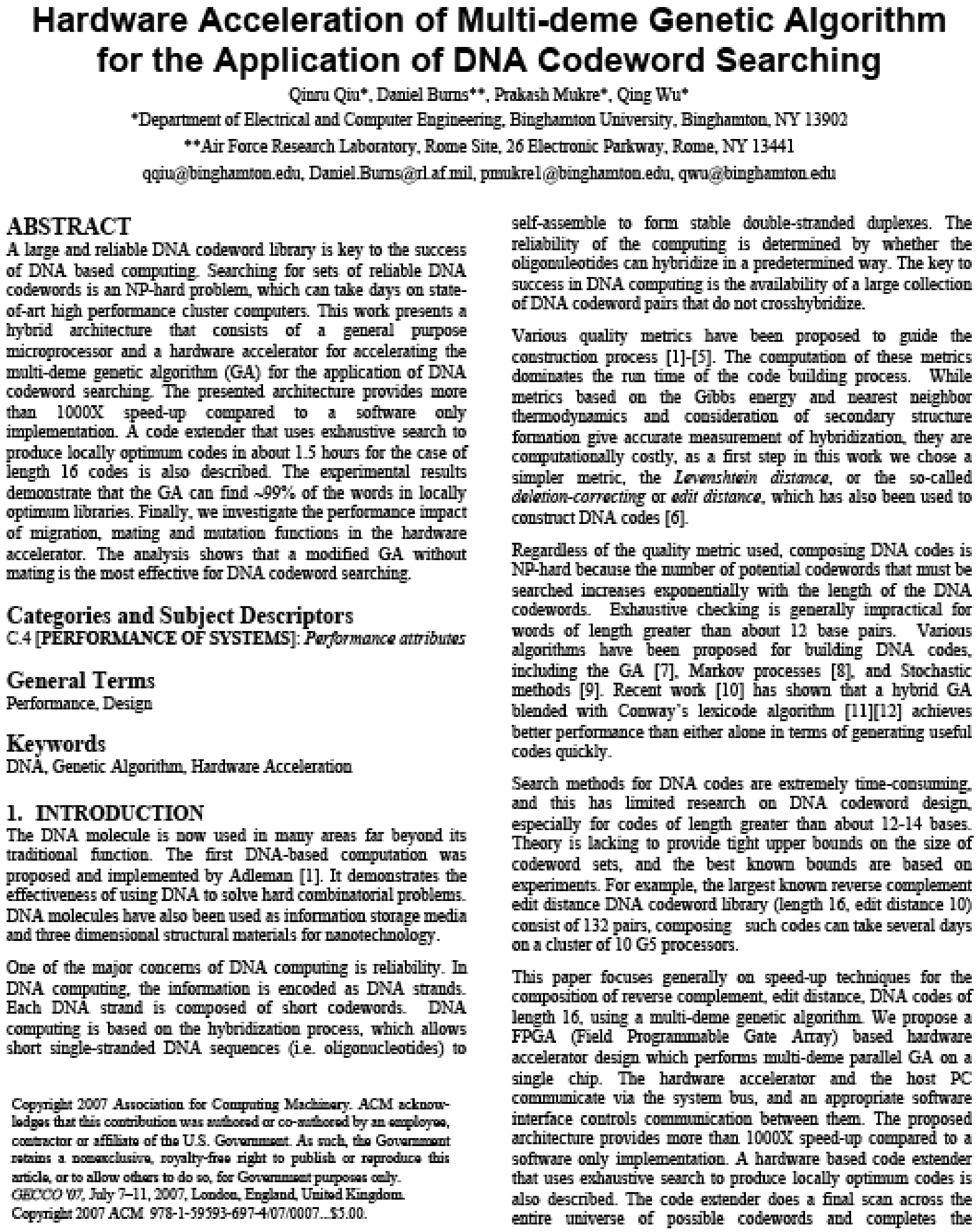
This paper focuses generally on speed-up techniques for the composition of reverse complement, edit distance, DNA codes of length 16, using a modified genetic algorithm that uses a locally exhaustive, mutation-only heuristic tuned for speed. Ongoing work to be reported elsewhere is addressing extensions to metrics involving nearest neighbor thermodynamics, a more general GA, codewords of length 32.

More specifically, we report a novel accelerator for DNA codeword composition that incorporates a hardware GA, hardware edit distance calculation, and hardware exhaustive search. Hardware exhaustive search extends an initial codeword library by doing a final scan across the entire universe of possible codewords, yielding a known locally optimum code. The proposed architecture consists of a host PC, a hardware accelerator implemented in reconfigurable logic on a *field programmable gate array* (FPGA) and a software program running in a host PC that controls and communicates with the hardware accelerator. The characteristics of the proposed architecture are as follows:

1. High performance. It utilizes programmable logic devices to enable pipelined and massively parallel processing of the data. Compared with software-only approaches, the new architecture can provide more than 1000X speed-up. For example, instead of 52 days, it only takes 1.5 hours to scan

1-4244-0710-9/07/$20.00 ©2007 IEEE

Appendix F. GECCO 2007 paper [Reference 27]. **(Click to view Appendix_F.pdf).**



Hardware Acceleration of Multi-deme Genetic Algorithm for the Application of DNA Codeword Searching

Qinru Qiu*, Daniel Burns**, Prakash Mukre*, Qing Wu*
*Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902
**Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441
qqiu@binghamton.edu, Daniel.Burns@rl.af.mil, pmukre1@binghamton.edu, qwu@binghamton.edu

ABSTRACT

A large and reliable DNA codeword library is key to the success of DNA based computing. Searching for sets of reliable DNA codewords is an NP-hard problem, which can take days on state-of-art high performance cluster computers. This work presents a hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the multi-deme genetic algorithm (GA) for the application of DNA codeword searching. The presented architecture provides more than 1000X speed-up compared to a software only implementation. A code extender that uses exhaustive search to produce locally optimum codes in about 1.5 hours for the case of length 16 codes is also described. The experimental results demonstrate that the GA can find ~99% of the words in locally optimum libraries. Finally, we investigate the performance impact of migration, mating and mutation functions in the hardware accelerator. The analysis shows that a modified GA without mating is the most effective for DNA codeword searching.

Categories and Subject Descriptors

C.4 [PERFORMANCE OF SYSTEMS]: *Performance attributes*

General Terms

Performance, Design

Keywords

DNA, Genetic Algorithm, Hardware Acceleration

1. INTRODUCTION

The DNA molecule is now used in many areas far beyond its traditional function. The first DNA-based computation was proposed and implemented by Adleman [1]. It demonstrates the effectiveness of using DNA to solve hard combinatorial problems. DNA molecules have also been used as information storage media and three dimensional structural materials for nanotechnology.

One of the major concerns of DNA computing is reliability. In DNA computing, the information is encoded as DNA strands. Each DNA strand is composed of short codewords. DNA computing is based on the hybridization process, which allows short single-stranded DNA sequences (i.e. oligonucleotides) to self-assemble to form stable double-stranded duplexes. The reliability of the computing is determined by whether the oligonuleotides can hybridize in a predetermined way. The key to success in DNA computing is the availability of a large collection of DNA codeword pairs that do not crosshybridize.

Various quality metrics have been proposed to guide the construction process [1]-[5]. The computation of these metrics dominates the run time of the code building process. While metrics based on the Gibbs energy and nearest neighbor thermodynamics and consideration of secondary structure formation give accurate measurement of hybridization, they are computationally costly, as a first step in this work we chose a simpler metric, the *Levenshtein distance*, or the so-called *deletion-correcting* or *edit distance*, which has also been used to construct DNA codes [6].

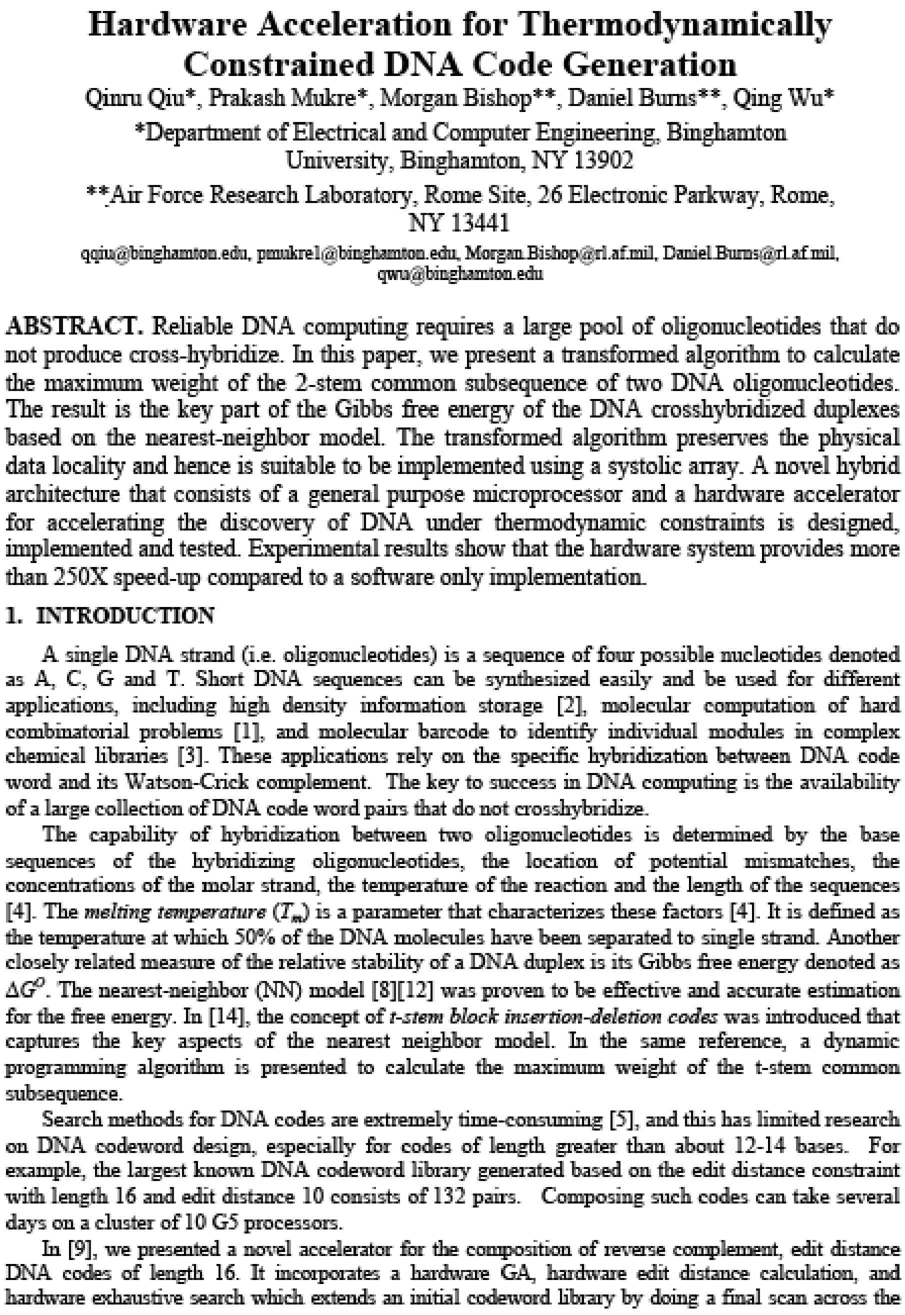
Regardless of the quality metric used, composing DNA codes is NP-hard because the number of potential codewords that must be searched increases exponentially with the length of the DNA codewords. Exhaustive checking is generally impractical for words of length greater than about 12 base pairs. Various algorithms have been proposed for building DNA codes, including the GA [7], Markov processes [8], and Stochastic methods [9]. Recent work [10] has shown that a hybrid GA blended with Conway's lexicode algorithm [11][12] achieves better performance than either alone in terms of generating useful codes quickly.

Search methods for DNA codes are extremely time-consuming, and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. Theory is lacking to provide tight upper bounds on the size of codeword sets, and the best known bounds are based on experiments. For example, the largest known reverse complement edit distance DNA codeword library (length 16, edit distance 10) consist of 132 pairs, composing such codes can take several days on a cluster of 10 G5 processors.

This paper focuses generally on speed-up techniques for the composition of reverse complement, edit distance, DNA codes of length 16, using a multi-deme genetic algorithm. We propose a FPGA (Field Programmable Gate Array) based hardware accelerator design which performs multi-deme parallel GA on a single chip. The hardware accelerator and the host PC communicate via the system bus, and an appropriate software interface controls communication between them. The proposed architecture provides more than 1000X speed-up compared to a software only implementation. A hardware based code extender that uses exhaustive search to produce locally optimum codes is also described. The code extender does a final scan across the entire universe of possible codewords and completes the

GECCO '07, July 7–11, 2007, London, England, United Kingdom.

Appendix G. DNA 2007 paper [Reference 28]. (Click to open Appendix G.)



Hardware Acceleration for Thermodynamically Constrained DNA Code Generation

Qinru Qiu*, Prakash Mukre*, Morgan Bishop**, Daniel Burns**, Qing Wu*

*Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902

**Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441

qqiu@binghamton.edu, pmukre1@binghamton.edu, Morgan.Bishop@rl.af.mil, Daniel.Burns@rl.af.mil, qwu@binghamton.edu

ABSTRACT. Reliable DNA computing requires a large pool of oligonucleotides that do not produce cross-hybridize. In this paper, we present a transformed algorithm to calculate the maximum weight of the 2-stem common subsequence of two DNA oligonucleotides. The result is the key part of the Gibbs free energy of the DNA crosshybridized duplexes based on the nearest-neighbor model. The transformed algorithm preserves the physical data locality and hence is suitable to be implemented using a systolic array. A novel hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the discovery of DNA under thermodynamic constraints is designed, implemented and tested. Experimental results show that the hardware system provides more than 250X speed-up compared to a software only implementation.

1. INTRODUCTION

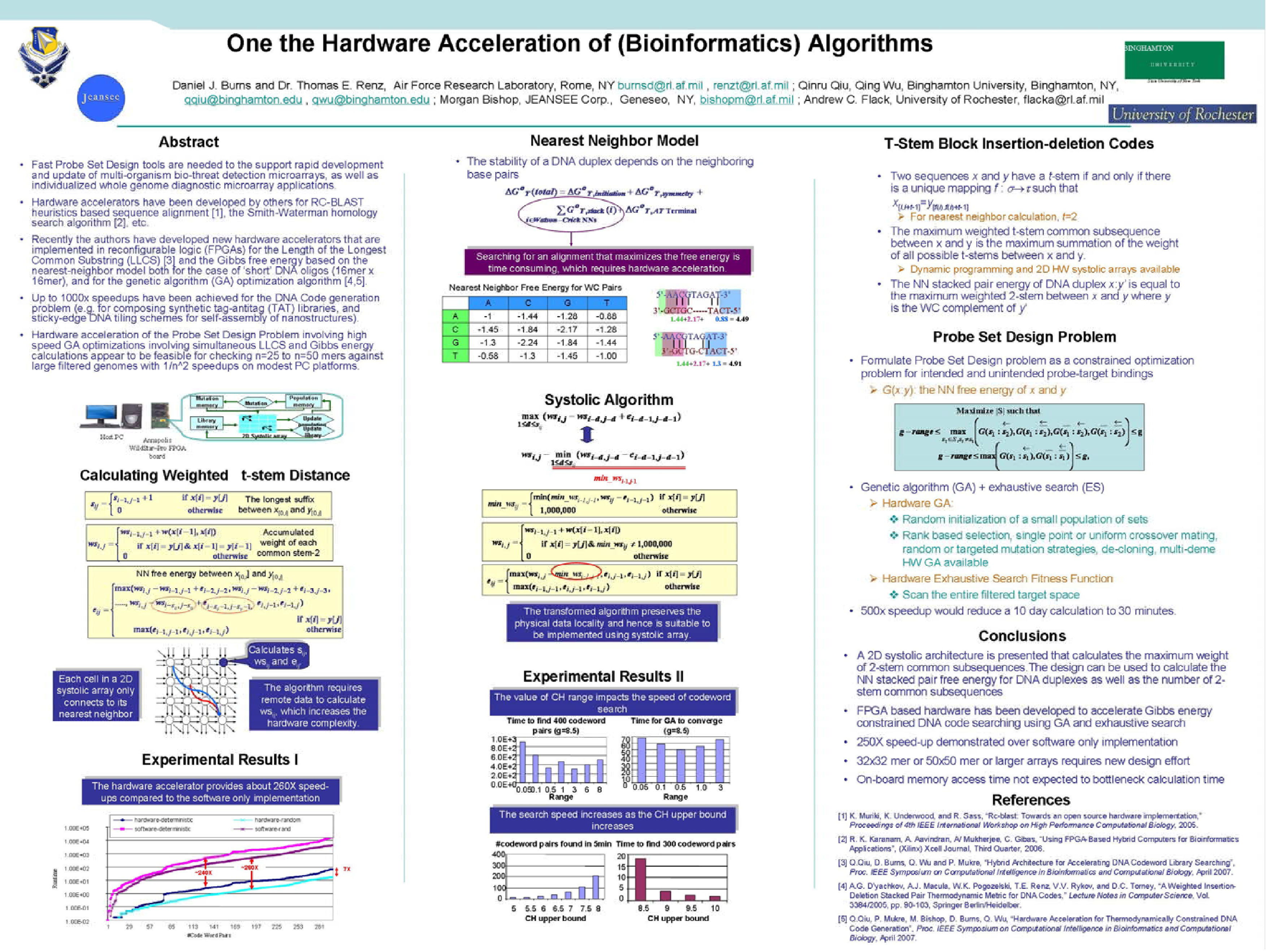
A single DNA strand (i.e. oligonucleotides) is a sequence of four possible nucleotides denoted as A, C, G and T. Short DNA sequences can be synthesized easily and be used for different applications, including high density information storage [2], molecular computation of hard combinatorial problems [1], and molecular barcode to identify individual modules in complex chemical libraries [3]. These applications rely on the specific hybridization between DNA code word and its Watson-Crick complement. The key to success in DNA computing is the availability of a large collection of DNA code word pairs that do not crosshybridize.

The capability of hybridization between two oligonucleotides is determined by the base sequences of the hybridizing oligonucleotides, the location of potential mismatches, the concentrations of the molar strand, the temperature of the reaction and the length of the sequences [4]. The *melting temperature* (T_m) is a parameter that characterizes these factors [4]. It is defined as the temperature at which 50% of the DNA molecules have been separated to single strand. Another closely related measure of the relative stability of a DNA duplex is its Gibbs free energy denoted as ΔG^O. The nearest-neighbor (NN) model [8][12] was proven to be effective and accurate estimation for the free energy. In [14], the concept of *t-stem block insertion-deletion codes* was introduced that captures the key aspects of the nearest neighbor model. In the same reference, a dynamic programming algorithm is presented to calculate the maximum weight of the t-stem common subsequence.

Search methods for DNA codes are extremely time-consuming [5], and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. For example, the largest known DNA codeword library generated based on the edit distance constraint with length 16 and edit distance 10 consists of 132 pairs. Composing such codes can take several days on a cluster of 10 G5 processors.

In [9], we presented a novel accelerator for the composition of reverse complement, edit distance DNA codes of length 16. It incorporates a hardware GA, hardware edit distance calculation, and hardware exhaustive search which extends an initial codeword library by doing a final scan across the

Appendix H. SIBD 2007 paper. [Reference 32]. (Click to view Appendix_H.pdf).



One the Hardware Acceleration of (Bioinformatics) Algorithms

Daniel J. Burns and Dr. Thomas E. Renz, Air Force Research Laboratory, Rome, NY burnsd@rl.af.mil , renzt@rl.af.mil ; Qinru Qiu, Qing Wu, Binghamton University, Binghamton, NY, qqiu@binghamton.edu , qwu@binghamton.edu ; Morgan Bishop, JEANSEE Corp., Geneseo, NY, bishopm@rl.af.mil ; Andrew C. Flack, University of Rochester, flacka@rl.af.mil

Abstract

- Fast Probe Set Design tools are needed to the support rapid development and update of multi-organism bio-threat detection microarrays, as well as individualized whole genome diagnostic microarray applications.
- Hardware accelerators have been developed by others for RC-BLAST heuristics based sequence alignment [1], the Smith-Waterman homology search algorithm [2], etc.
- Recently the authors have developed new hardware accelerators that are implemented in reconfigurable logic (FPGAs) for the Length of the Longest Common Substring (LLCS) [3] and the Gibbs free energy based on the nearest-neighbor model both for the case of 'short' DNA oligos (16mer x 16mer), and for the genetic algorithm (GA) optimization algorithm [4,5].
- Up to 1000x speedups have been achieved for the DNA Code generation problem (e.g. for composing synthetic tag-antitag (TAT) libraries, and sticky-edge DNA tiling schemes for self-assembly of nanostructures).
- Hardware acceleration of the Probe Set Design Problem involving high speed GA optimizations involving simultaneous LLCS and Gibbs energy calculations appear to be feasible for checking n=25 to n=50 mers against large filtered genomes with 1/n^2 speedups on modest PC platforms.

Calculating Weighted t-stem Distance

$$s_{ij} = \begin{cases} s_{i-1,j-1}+1 & \text{if } x[i]=y[j] \\ 0 & \text{otherwise} \end{cases}$$

The longest suffix between $x_{[0,i]}$ and $y_{[0,j]}$

$$ws_{i,j} = \begin{cases} ws_{i-1,j-1} + w(x[i-1], x[i]) \\ \quad \text{if } x[i]=y[j] \,\&\, x[i-1]=y[i-1] \\ 0 \quad \text{otherwise} \end{cases}$$

Accumulated weight of each common stem-2

NN free energy between $x_{[0,i]}$ and $y_{[0,j]}$

$$e_{ij} = \begin{cases} \max(ws_{i,j} - ws_{i-1,j-1} + e_{i-2,j-2}, ws_{i,j} - ws_{i-2,j-2} + e_{i-3,j-3}, \\ \quad \ldots, ws_{i,j} - ws_{i-r_{ij},j-r_{ij}} + e_{i-r_{ij}-1,j-r_{ij}-1}, e_{i,j-1}, e_{i-1,j}) \\ \qquad \text{if } x[i]=y[j] \\ \max(e_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) \quad \text{otherwise} \end{cases}$$

Calculates s_{ij}, ws_{ij} and e_{ij}

Each cell in a 2D systolic array only connects to its nearest neighbor

The algorithm requires remote data to calculate ws_{ij}, which increases the hardware complexity.

Experimental Results I

The hardware accelerator provides about 260X speed-ups compared to the software only implementation

Nearest Neighbor Model

- The stability of a DNA duplex depends on the neighboring base pairs

$$\Delta G^{\circ}_{T}(total) = \Delta G^{\circ}_{T,initiation} + \Delta G^{\circ}_{T,symmetry} + \sum_{i \in Watson-Crick\ NNs} G^{\circ}_{T,stack}(i) + \Delta G^{\circ}_{T,AT\ Terminal}$$

Searching for an alignment that maximizes the free energy is time consuming, which requires hardware acceleration.

Nearest Neighbor Free Energy for WC Pairs

	A	C	G	T
A	-1	-1.44	-1.28	-0.88
C	-1.45	-1.84	-2.17	-1.28
G	-1.3	-2.24	-1.84	-1.44
T	-0.58	-1.3	-1.45	-1.00

Systolic Algorithm

$$\max_{1\le d\le r_{ij}} (ws_{i,j} - ws_{i-d,j-d} + e_{i-d-1,j-d-1})$$

$$\Updownarrow$$

$$ws_{i,j} - \underline{\min_{1\le d\le r_{ij}} (ws_{i-d,j-d} - e_{i-d-1,j-d-1})}$$

$min\_ws_{i-1,j-1}$

$$min\_ws_{ij} = \begin{cases} \min(min\_ws_{i-1,j-1}, ws_{ij} - e_{i-1,j-1}) & \text{if } x[i]=y[j] \\ 1{,}000{,}000 & \text{otherwise} \end{cases}$$

$$ws_{i,j} = \begin{cases} ws_{i-1,j-1} + w(x[i-1], x[i]) \\ \quad \text{if } x[i]=y[j] \,\&\, min\_ws_{ij} \ne 1{,}000{,}000 \\ 0 \quad \text{otherwise} \end{cases}$$

$$e_{ij} = \begin{cases} \max(ws_{i,j} - min\_ws_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) & \text{if } x[i]=y[j] \\ \max(e_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) & \text{otherwise} \end{cases}$$

The transformed algorithm preserves the physical data locality and hence is suitable to be implemented using systolic array.

Experimental Results II

The value of CH range impacts the speed of codeword search

The search speed increases as the CH upper bound increases

T-Stem Block Insertion-deletion Codes

- Two sequences *x* and *y* have a *t*-stem if and only if there is a unique mapping $f : \sigma \rightarrow \tau$ such that $x_{[i,i+t-1]} = y_{[f(i),f(i)+t-1]}$
  - For nearest neighbor calculation, *t*=2
- The maximum weighted t-stem common subsequence between x and y is the maximum summation of the weight of all possible t-stems between x and y.
  - Dynamic programming and 2D HW systolic arrays available
- The NN stacked pair energy of DNA duplex *x*:*y'* is equal to the maximum weighted 2-stem between *x* and *y* where *y* is the WC complement of *y'*

Probe Set Design Problem

- Formulate Probe Set Design problem as a constrained optimization problem for intended and unintended probe-target bindings
  - *G*(*x*:*y*): the NN free energy of *x* and *y*

Maximize |S| such that

$$g - range \le \max_{s_1 \in S, s_1 \ne s_2} \left(G(s_1 : \overleftarrow{s_2}), G(s_1 : \overleftarrow{\overline{s_2}}), G(\overline{s_1} : \overleftarrow{s_2}), G(\overline{s_1} : \overleftarrow{\overline{s_2}}) \right) \le g$$

$$g - range \le \max \left(G(s_1 : \overleftarrow{s_1}), G(\overline{s_1} : \overleftarrow{\overline{s_1}}) \right) \le g,$$

- Genetic algorithm (GA) + exhaustive search (ES)
  - Hardware GA:
    - Random initialization of a small population of sets
    - Rank based selection, single point or uniform crossover mating, random or targeted mutation strategies, de-cloning, multi-deme HW GA available
  - Hardware Exhaustive Search Fitness Function
    - Scan the entire filtered target space
- 500x speedup would reduce a 10 day calculation to 30 minutes.

Conclusions

- A 2D systolic architecture is presented that calculates the maximum weight of 2-stem common subsequences.The design can be used to calculate the NN stacked pair free energy for DNA duplexes as well as the number of 2-stem common subsequences
- FPGA based hardware has been developed to accelerate Gibbs energy constrained DNA code searching using GA and exhaustive search
- 250X speed-up demonstrated over software only implementation
- 32x32 mer or 50x50 mer or larger arrays requires new design effort
- On-board memory access time not expected to bottleneck calculation time

References

[1] K. Muriki, K. Underwood, and R. Sass, "Rc-blast: Towards an open source hardware implementation," *Proceedings of 4th IEEE International Workshop on High Performance Computational Biology*, 2005.

[2] R. K. Karanam, A. Aavindran, A/ Mukherjee, C. Gibas, "Using FPGA-Based Hybrid Computers for Bioinformatics Applications", (Xilinx) Xcell Journal, Third Quarter, 2006.

[3] Q.Qiu, D. Burns, Q. Wu and P. Mukre, "Hybrid Architecture for Accelerating DNA Codeword Library Searching", *Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, April 2007.

[4] A.G. D'yachkov, A.J. Macula, W.K. Pogozelski, T.E. Renz, V.V. Rykov, and D.C. Torney, "A Weighted Insertion-Deletion Stacked Pair Thermodynamic Metric for DNA Codes," *Lecture Notes in Computer Science*, Vol. 3384/2005, pp. 90-103, Springer Berlin/Heidelber.

[5] Q.Qiu, P. Mukre, M. Bishop, D. Burns, Q. Wu, "Hardware Acceleration for Thermodynamically Constrained DNA Code Generation", *Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, April 2007.

Lifetime Aware Resource Management for Sensor Network Using Distributed Genetic Algorithm

Qinru Qiu Qing Wu
Department of Electrical and Computer Engineering
Binghamton University
Binghamton, NY 13902
001-607-777-4918, 001-607-777-4536
{qqiu, qwu}@binghamton.edu

Daniel Burns Douglas Holzhauer
Air Force Research Laboratory, Rome Site
26 Electronic Parkway
Rome, NY 13441
001-315-330-2335, 001-315-330-4920
{Daniel.Burns, Douglas.Holzhauer}@rl.af.mil

ABSTRACT

In this work we consider lifetime-aware resource management for sensor network using distributed genetic algorithm (GA). Our goal is to allocate different detection methods to different sensor nodes in the way such that the required detection probability can be achieved while the network lifetime is maximized. The contribution of this paper is twofold. Firstly, the resource management problem is formulated as a constraint optimization problem and is solved using a distributed GA. Secondly, empirical analysis results are provided that reveals the relationship between the configuration parameters and the quality of the search. A regression model is designed to estimate the runtime of the distributed GA given the configuration parameters. The model is utilized to find energy efficient configurations of the algorithm.

Categories and Subject Descriptors

J.7 [**Computer Applications**]: Sensors and Sensor Networks

General Terms

Experimentation

Keywords

Distributed Genetic Algorithm, Sensor Network, Energy Aware Design, Resource Management

1. INTRODUCTION

Due to the fast development of information technology, the networked distributed system is gradually replacing the conventional centralized system. It is a vision of the future that large numbers of low cost smart mobile devices will be integrated into the daily life of ordinary people. Accumulated, they provide the information processing capability that is equivalent to a high performance processing station. The emerging concept of Ambient Intelligence [1] and the recent developments of sensor networks [2], and wearable computers [3] reflect such vision. A distributed system consists of multiple heterogeneous networked processing elements, which are battery-powered and work on a set of tasks collaboratively. Each processing element has limited resources, such as battery energy, communication bandwidth, etc. It is a challenging task to efficiently utilize these resources to deliver required services during the runtime in a dynamic environment.

Resource management is defined as the process that assigns tasks to different processing elements, schedules their start times and decides the level of service quality, which determines the resource usage, such as the energy dissipation and communication bandwidth, to run these tasks. The execution of each task represents a positive gain when measuring or quantifying the performance of the system. It also associates a cost, which represents the resource usage. The resource management problem can be formulated as a multi-objective optimization problem, i.e. maximizing the gain while minimizing the cost. It can also be formulated as a constraint optimization problem, i.e. maximizing the gain while satisfying the cost constraint or vise versa.

In this paper we focus on the management of the energy resource in an environment monitoring sensor network that is used to monitor, model and forecast physical processes, such as environment pollution, flooding, and fire etc. The basic configuration of each node in this network consists of a microprocessor, a wireless transceiver and an array of sensors such as light detector, barometer, humidity and thermopile sensors. A set of data acquisition and signal processing applications is available on each node. They provide the tradeoffs between detection quality and resource utilization. For example, increasing the sampling rate improves the probability of detecting an abnormal event however it increases the power consumption as well.

There is usually a significant cost associated with deploying an environment monitoring system. It is desirable that the system can work for a reasonably long time after it is deployed. A common approach is to incorporate certain level of redundancy in the system. More than one node usually will be deployed to cover the same region. These nodes may be turned on alternatively to extend the network lifetime or simultaneously to increase the detection probability. If the minimum detection accuracy is given as a user constraint, the resource management problem for the system is to determine which sensor nodes should be turned on to process which data acquisition and signal processing application such that the network lifetime can be maximized while meeting the required detection accuracy. This is a well known general assignment problem which has been proven to be NP-complete [4].

Most of the traditional resource optimization algorithms are solved in a centralized, off-line approach which is not suitable for a distributed system. In this paper we study the use of distributed genetic algorithm (GA) to solve the above mentioned optimization problem, potentially using processing capabilities residing on nodes of the distributed sensor network. One of the major characteristics of the GA is that it is "embarrassingly parallel", in the sense that, its workload can easily be evenly distributed among processors, making it an appropriate choice for solving optimization problems

ISLPED'06, October 4–6, 2006, Tegernsee, Germany.
Copyright 2006 ACM 1-59593-462-6/06/0010...$5.00.

in distributed systems. The configurations of the distributed, multi-deme GA, such as the population size, the migration rate, and the parallelism, has a significant impact on the quality of the search [8]. Finding efficient configurations of the distributed GA is an important research topic. The contribution of this paper is twofold. Firstly, the resource management problem is formulated as a constraint optimization problem and is solved using a distributed GA. The simulation results show that the resulting task allocation scheme increases the system lifetime by 14.4% in average, comparing to heuristic approaches. Secondly, empirical analysis results are provided that reveals the relationship between the configuration parameters and the quality of the search. A regression model is presented that estimates the runtime of the distributed GA given the configuration parameters. This model is then used to find energy efficient configurations of the algorithm.

Many previous works on sensor network resource management and task allocation address network communication issues [5][6]. In these schemes the nodes are dynamically awakened to route a message. In reference [7] the resource allocation problem in a vehicle tracking system is modeled as a virtual market and solved using feedback control. This work focuses more on the tracking of a moving object rather than the collaborative detection of a static event. Therefore it cannot be applied in the environment monitoring system. Reference [9] focuses on task allocation on the gateways in a cluster-based sensor network. The problem is also formulated as a constraint optimization problem and is solved using simulated annealing, which is a centralized stochastic searching algorithm. Compared with reference [9], the resource management problem considered in this paper has a different set of constraints and objective functions and is solved using a distributed GA.

The rest of this paper is organized as follows. Section 2 introduces the sensor network architecture. Section 3 presents the distributed GA algorithm. Section 4 provides the empirical analysis of the relationship between the configuration parameters and the quality of search of GA and derives the regression model for runtime estimation. Section 5 discusses the utilization of the regression model to design energy efficient distributed GA. Sections 6 and 7 provide the experimental results and summaries, respectively.

2. SENSOR NETWORK ARCHITECTURE

We consider the sensor network that is deployed with a certain level of redundancy. The network can be partitioned into several clusters. Each cluster consists of *p* sensor nodes that are responsible for performing monitoring and hazard detection in the same region. Each sensor node is low cost and low quality; however combined together they provide very accurate detection. The nodes in the same cluster have direct communication with each other via wireless communication channels. The nodes in different clusters communicate with each other through gateways. In this work we assume that the clustering and routing scheme is provided. We also assume that each cluster has advanced data fusion capability so that the traffic of inter-cluster communication is low.

An array of *w* sensors is installed on each node. The reading from these sensors can be sampled by *l* different sampling frequencies. Obviously, higher sampling frequency leads to higher detection probability while consumes more energy. The sampled data from sensor *i* can be analyzed in x_i different ways. They provide different tradeoffs between accuracy and energy dissipation. A *detection method* (i.e. *task*) is considered as a combination of sensing function, sampling frequency and signal processing algorithm.

Each task-processor pair (*i*, *k*), $1 \leq i \leq n$ and $1 \leq k \leq p$, associates with two variables $pow_{i,k}$ and $prob_{i,k}$, which represent the power consumption and the detection probability of task *i* when it is running on processor *k*. The $prob_{i,k}$ is a function of the location and the environment of the sensor node. We assume that this function is pre-calibrated and installed on each sensor node before its deployment. The sensor node will collect the environment information and calculate the detection probability using the provided function periodically. To improve the detection probability, the node is allowed to use more than one detection method at the same time. The combined detection probability of node *k* can be calculated as $1-\prod_{i\in\Delta_k}(1-prob_{i,k})$, where Δ_k is the set of tasks that are allocated to node *k*. The total node power consumption can be calculated as $\sum_{i\in\Delta_k} pow_{i,k}$. The detection probability *Prob* of a cluster with *p* nodes is calculated as $Prob = 1-\prod_{1\leq k\leq p}\prod_{i\in\Delta_k}(1-prob_{i,k})$.

The goal of resource management is to find the Δ_k for each processor *k* so that the combined detection probability of the cluster is larger than the user defined constraint while the network lifetime is maximized. In this work, we define the network lifetime as the time from the deployment of the sensor network to the time when the first node runs out of battery energy. We assume that each sensor node is built with the smart battery Bus (SMBus) [10] which enables the system software to keep tracking of the remaining battery capacity and estimate the remaining lifetime.

3. RESOURCE MANAGEMENT USING DISTRIBUTED GA

A Genetic Algorithm (GA) is a stochastic search technique based on the mechanism of natural selection and recombination. It starts with an initial *population* of individuals, i.e. a set of randomly generated candidate solutions. The solutions are represented by *chromosomes*, which are collections of numbers or symbols that map onto parameters of the problem. Individuals are evolved from generation to generation, with *selection*, *mating*, and *mutation* operators that provide an effective combination of exploration of the global search space and pressure to converge to the global minimum. The solution quality is measured by a *fitness* function.

The Island multi-deme GA is one of the parallel GA models that are widely used [8]. In this model, the population is divided into several sub-populations and distributed on different processors. Each sub-population evolves independently for a few generations, before one or more of the best individuals of the sub-populations migrate across processors. The time between migrations is called *epoch*.

In this work the Island multi-deme GA is used to optimize the resource management for a cluster of sensor nodes. Each individual solution is a chromosome of *n* symbols, where *n* is the total number of tasks in the cluster. We assume that each task can only be selected by at most one sensor node in a cluster because multiple executions of the same task only generate redundant information. If the *j*th task is allocated to node *x* then the *j*th entry of the chromosome is equal to *x*. If the *j*th entry of the chromosome is -1 then this task is not allocated to any of the processors. Denote the user specified minimum detection probability as $prob_{th}$, the fitness function is:

$$fitness = \begin{cases} 0 & \text{if } Prob < Prob_{th} \\ \min\limits_{1\leq k\leq p}(B_k / \sum\limits_{i\in\Delta_k} pow_{i,k}) & \text{otherwise} \end{cases} \tag{1}$$

where B_k is the remaining battery capacity of node *k*. The fitness of an individual is 0 if the corresponding resource management scheme cannot meet the user specified detection threshold; otherwise its fitness is equal to the minimum remaining lifetime of the nodes. Single point crossover mating function is used in our experiment. The mutation probability is set to 1%, and involves flipping bits in integer representations of the parameters stored in chromosomes.

The GA is running on *np* processors. Each sub-population is initialized randomly and its size is denoted as *pop*. The sub-population evolves independently for *c* generations, and then 5 of the best individuals are broadcast to all other processors. The three parameters, *np*, *pop* and *c*, will be referred as the *configuration parameters* in the rest of this paper. The value of the configuration parameters has significant impact on the convergence speed of the GA and the quality of the solution. An empirical analysis is next.

4. CONFIGURATION PARAMETERS

We are interested in understanding the effect of configuration parameters on the quality of the search of the distributed GA that is previously discussed. Some work has been carried out in this area [8]. However, most of these involve the analysis of simple optimization problems such as the fully deceptive function [12]. Whether their results can be applied to our problem is unknown. Due to the extremely large search space and very complicated stochastic behavior of the GA, we found that it is difficult to perform an analytical study. Therefore, extensive experiments have been simulated and the relation between the configuration parameters and the quality of search is derived empirically.

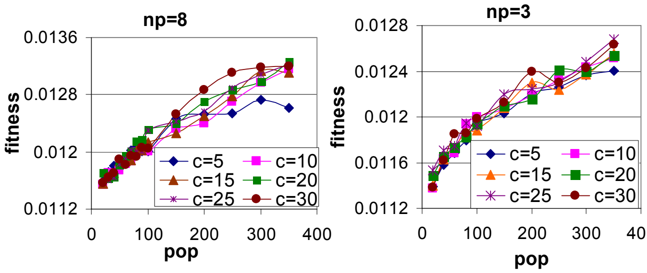
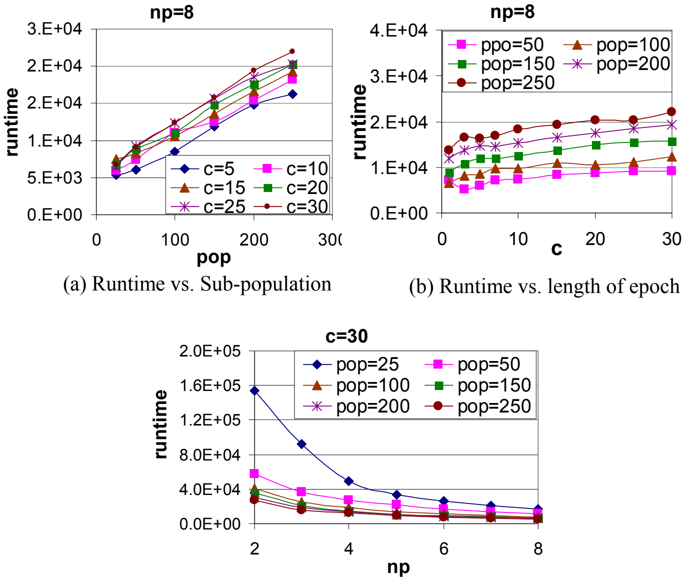


Figure 1 Normalized fitness vs. Sub-population size

Two sets of experiments have been carried out. In these experiments, we model a cluster of 10 sensor nodes. There are 100 tasks available. The GA is running on *np* sensor nodes with *np* ≤ 10. The detection probability and the power consumption of each task are uniformly distributed random variables whose range is 1% ~ 25% and 0.1Watt ~ 10Watt respectively. The battery of each sensor has the capacity of 5000 Ampere·hour and the V_{dd} is 1V. Because GA is a stochastic algorithm, we run each simulation 50 times and report the mean value.

The first set of experiments is designed to find out the effect of the configuration parameters on the quality of the solution. We swept the *np* from 2 to 8, the *pop* from 25 to 350, and the *c* from 1 to 35. For each configuration, the distributed GA is simulated. The GA will stop when the fitness of the best individual does not improve for 2000 generations. The relation between the *pop* and the normalized fitness of the best individual is reported. Figure 1 shows two sets of data for *np* (i.e. the number of processors) equal to 8 and 3. The results show that increasing both the *pop* and the *np* improves the quality of the solution. However, varying *c* has very little impact on it. Therefore the quality of the solution is determined by the size of the total population which is the product of the *pop* and *np*.

The second set of experiments is designed to find out the effect of the configuration parameters on the runtime of the GA. The value of *np*, *pop* and *c* are swept in the same way as the first experiment. The GA stops when the fitness of the best individual exceeds the threshold which is set to be 5 times of the expected fitness of a random individual. The number of generations that the GA has iterated is reported. Due to the iterative nature of GA, it is reasonable to assume that the runtime of each generation is approximately the same and it increases linearly as population size increases. Therefore we use the product of *pop* and the number of generations that the GA has iterated as a measure of the runtime.



(a) Runtime vs. Sub-population

(b) Runtime vs. length of epoch

(c) Runtime vs. parallelism

Figure 2 Runtime vs. configuration parameters

The relation among *pop*, *c*, *np* and the runtime are extracted from the results of second experiment. Figure 2 (a)-(c) show some of the data that we have obtained. Several observations can be made from these data. First, when the size of the sub-population increases, the runtime increases linearly. Combined with the results from the first experiment we can see that if the goal of the GA is to find the best possible solution, then a large population should be used. However, if the goal of the GA is to find a good solution in a short time, then increasing the population size will not help. Instead a small population should be used. Second, reducing the migration rate will result an almost linear increase in solution time. The slope is the same for different sub population size. Third, increasing the number of processors will reduce runtime, and this effect is more dominant when the sub-population is small.

In order to consider the combined effect of all of the three configuration parameters, we introduce a new variable called *effective population* (*Epop*). The size of the effective population increases when the size of the sub-population, the parallelism or the migration rate increases. It can be calculated as the following:

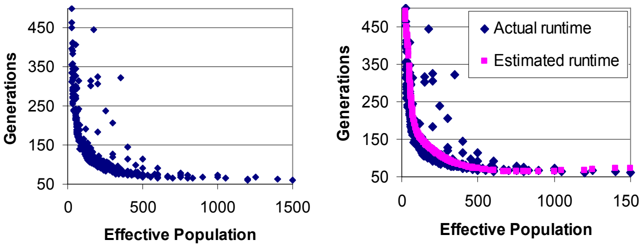
$$Epop = pop + pop \cdot (np-1)/c \qquad (2)$$

Given the effective population, the runtime of the distributed GA can be predicted. Let *G* denote the number of generations that the GA has iterated before it finds the solution with the required fitness. Figure 3 (a) gives the relation between *Epop* and *G*. It shows that *G* is a continuous and differentiable function of *Epop*.

Based on the observation, we construct a prediction model to predict the number of generations that the GA has iterated.

$$G = a + a_0 / \sqrt{Epop} + \sum_{i=1}^{5} a_i / Epop^i \qquad (3)$$

The coefficients a, a_0 …, a_5 are obtained using regression analysis. Note that the value of the coefficients will change if the experiment setup changes. Here the experiment setup includes the threshold of fitness and the distribution function of *prob* and *power*. For each new setup, regression analysis should be performed to obtain the values of the coefficients.



(a) G vs. effective population G (b) Comparing predicted and actual G

Figure 3 Runtime vs. effective population

The above model gives quite accurate prediction of the number of generations that GA has iterated given the configuration parameters. Figure 3 (b) compares the prediction model with the simulated results. The blue dots give the G value obtained from simulation and the magenta dots give the G value obtained using the prediction model. The runtime T is measured as the product of *pop* and the number of generations $T = G \cdot pop$.

5. ENERGY EFFICIENT CONFIGURATIONS OF DISTRIBUTED GA

Sensor nodes are energy constraint systems. Any application running on the sensor node should be designed carefully to achieve high speed and energy efficiency. In this section we will discuss how to select the configuration parameters to minimize the energy dissipation of the distributed GA.

In a computing system with fixed supply voltage (V_{dd}) and clock frequency, reducing the runtime of an algorithm leads to linear reduction of the energy dissipation if the processor can be turned off after the program finishes. More energy saving is possible by using *dynamic voltage and frequency scaling* (*DVFS*), which is one of the runtime power management approaches that is supported by many processors for the state-of-the-art mobile computing platforms. It is a property of CMOS digital circuit that reducing the V_{dd} can reduce the energy dissipation quadratically but increase the circuit delay linearly [11]. In a system with DVFS capability, the program is running at the minimum supply voltage and clock frequency so that it finishes just before the deadline. Due to the convex relation between the energy and the runtime, this gives more energy saving than running the program at the nominal speed and turn off the processor after the program finishes.

Population migration among the processors is an important feature in distributed GA. The communication energy to broadcast the best individuals must be considered. Under the assumption of a fixed transmission power and a constant transmission speed, the communication energy is proportional to the size of the transmitted data. The communication energy will not be affected by DVFS.

The computing energy is a product of the runtime and the power consumption of the processor. Therefore, the runtime model proposed in Section 4 is the key for the energy estimation of the distributed GA. While increasing the migration rate, decreasing the population size and increasing the parallelism reduce the runtime of GA and consequently reduce the computing energy, frequent population migration leads to high communication energy. The configuration parameters must be selected carefully to minimize the overall system energy dissipation, which is the sum of computing energy and communication energy.

Let T_{nom} denote the process time for a single individual in each generation at nominal V_{dd} and let p_{nom} denote the power consumption of the processor at nominal V_{dd}. The energy dissipation of GA on a processor without DVFS can be calculated as:

$$E = T_{nom} \cdot p_{nom} \cdot T + G / c \cdot N \cdot E_{bit} , \qquad (4)$$

where G is the number of generations that GA has iterated, T is the runtime of the GA that is measured as the product of *pop* and G, c is the length of an epoch, N is the size of data that is broadcasted during each population migration, and E_{bit} is the energy to transmit one bit data. The first term in equation (4) is the computing energy and the second term is the communication energy. Furthermore, $T_{nom} \cdot p_{nom}$ represents the computing energy to processor one individual in each generation and $N \cdot E_{bit}$ represents the communication energy to broadcast the best individual during one migration. T_{nom}, p_{nom}, and E_{bit} are hardware related constant parameters. N is determined by the size of migrations which is also a constant value. Because we are not interested in calculating the absolute energy dissipation, we simplify equation (4) and consider a normalized energy dissipation which is calculated as the following,

$$E_{norm} = \frac{E}{T_{nom} \cdot p_{nom}} = T + \frac{G}{c} E_{mig} , \qquad (5)$$

where E_{mig} is the ratio of communication energy versus computing energy and it is calculated as $E_{mig} = \frac{N \cdot E_{bit}}{T_{nom} \cdot p_{nom}}$. As we can see the value of E_{mig} is determined by the system hardware configuration. For example, the power consumption of a Lucent ORiNOCO USB Wireless Adapter is 360mA in TX mode and 245mA in RX mode. The typical active power of an Intel XScale processor is 300mA. Assume that the data is transmitted at 1Mbit/s. If N equals to 1k bytes and T_{nom} equals to 5μs, which is the time to run 10k instructions at 200MHz clock, then E_{mig} is approximately 160.

E_{norm} is an increasing function of *pop* and a decreasing function of *np* because changing these two parameters only affects the computing energy. The only configuration parameter that affects both the computing and communication energy is c. Provided with the value of *pop*, *np*, E_{mig}, it is not difficult to find the optimal c that minimizes E_{norm} by solving the differential equation $\frac{\partial E_{norm}}{\partial c} = 0$.

Because GA is running on multiple sensor nodes, the total energy dissipation can be calculated as $E_{total} = np \cdot E_{norm}$ where *np* is the parallelism of the GA.

If the DVFS is available on the processor, then the computing energy can be scaled quadratically as the runtime decreases. The energy dissipation of GA on each processor can be calculated as the following,

$$EDVFS = T_{nom} \cdot p_{nom} \cdot T \cdot s^2 + G / c \cdot N \cdot E_{bit} \qquad (6)$$

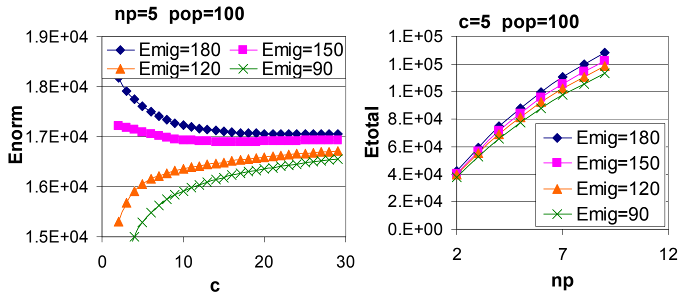
Here s is the scaling factor and it is calculated as $s = \frac{T_{nom} \cdot T}{T_{req}}$, where T_{req} is the deadline before which the GA must return a solution with the required fitness. Again, we simplify equation (6) and consider the normalized energy dissipation as the following,

$$EDVFS_{norm} = (T \cdot T_{nom} / T_{req})^3 + G / c \cdot E'_{mig}\,, \qquad (7)$$

E'_{mig} is calculated as $E'_{mig} = \dfrac{N \cdot E_{bit}}{p_{nom} \cdot T_{req}}$ which stands for the ratio of the communication energy for one migration versus the computing energy of the program if if takes exactly T_{req} time when running at the nominal V_{dd}. For the previous mentioned hardware system, which consists of Lucent ORiNOCO USB Wireless Adapter and Intel XScale processor, if the T_{req} is 1ms then E'_{mig} is approximately 0.8.

Again, the total energy dissipation can be calculated as $EDVFS_{total} = EDVFS_{norm} \cdot np$ and the optimal c that minimizes the energy dissipation can be found by solving the differential equation $\dfrac{\partial EDVFS_{norm}}{\partial c} = 0$.

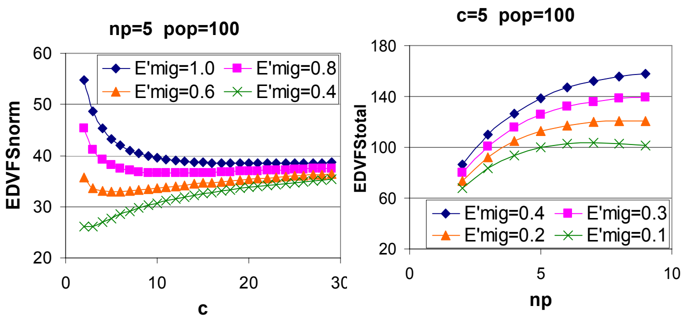
Plug in the runtime estimation of G and T into equation (4)~(7), the energy dissipation of GA can be expressed as a function of the configuration parameters. Figure 4 (a) shows the relation between E_{norm} and c in a system without DVFS. The *np* and *pop* are set to 5 and 100 respectively. The E_{mig} varies from 90 to 180. As we can see from the figure, the energy is an increasing function of c for small E_{mig} and a decreasing function of c for large E_{mig}. Furthermore, when the E_{mig} falls into certain range, the energy is first a decreasing then an increasing function of c. In this case, we need to solve the previous mentioned differential equation to find the most energy efficient migration rate. When the parameter c gets larger, the E_{norm} under different E_{mig} approach to the same value. This is because the migration rate is so low that a small difference in the communication energy does not have a significant affect on the total energy.



(a) E_{norm} vs. c

(b) E_{total} vs. np

Figure 4 Energy vs. configuration parameters without DVFS



(a) $EDVFS_{norm}$ vs. c

(b) $EDVFS_{total}$ vs. np

Figure 5 Energy vs. configuration parameters with DVFS

Figure 4 (b) shows the relation between E_{total} and np in a system without DVFS. The parameters c and *pop* are set to 5 and 100 respectively. E_{mig} varies from 90 to 180. It is interesting to note that the total energy always increases no matter how we change the E_{mig}. This indicates that without DVFS the energy efficiency will decrease as the parallelism increases.

Figure 5 (a) shows the relation between $EDVFS_{norm}$ and c in a system with DVFS. The *np* is set to 5, the *pop* is set to 100 and the E'_{mig} varies from 1.0 to 0.4. In this figure, we see the similar trend as what has been shown for the system without DVFS. Figure 5 (b) shows the relation between the $EDVFS_{total}$ and *np* with E'_{mig} varies from 0.4 to 0.1. As we can see that for systems with $E'_{mig} \geq 0.3$, increasing the parallelism always increases the total energy dissipation. However, for systems with $E'_{mig} < 0.3$, increasing the parallelism will first increase then decrease the total energy. This is because increasing the parallelism reduces the overall computing energy quadratically and increases the overall communication energy linearly. Eventually the quadratic decreasing in computing energy will become dominant.

6. EXPERIMENTAL RESULTS

In order to evaluate the performance of the GA based resource management scheme, a C++ based software program is constructed to emulate the environment monitoring sensor network. The cluster consists of 10 low cost and low quality sensor nodes and 100 tasks. The battery of each sensor has the capacity of 5000 Ampere·hour. The detection probability and the power consumption of each task are randomly generated. Different distributions with different variances are tested in the experiment. Furthermore, to emulate the behavior of the real sensor network which is deployed in a dynamic environment, the detection probability of the sensors is constantly changing. Every 1000 hours, for a set of x sensor nodes, their detection probability $prob_i$, $1 \leq i \leq 100$ will be regenerated and reapplied to model the change of their environment. The x is set to be 1, 2, and 5.

The environment setup is named by a quintuplet (*distribution*, *prob variance*, *power variance*, *biased/unbiased*, *x*). The first field specifies the type of distribution that is used to generate the detection probability and power consumption of each task. It can either be uniform distribution or normal distribution. The second and third filed specifies the variance of the detection probability and the power consumption respectively. The fourth field is either biased or unbiased. When an environment setup is biased, half of the sensor nodes have lower power consumption than the others. This field is designed to model a heterogeneous network. The final field specifies the number of sensors whose detection probability changes due to changes in the environment. Table 1 column 1 gives the list of environment setups that were tested in our experiments. Note that the variance of power consumption is different for the biased and unbiased environment.

Our distributed GA algorithm which is presented in section 4 is denoted as *GA-lifetime*, since its objective is to maximize the lifetime of the sensor network. The program is distributed on 5 processors ($np = 5$). The subpopulation size is set to 100 ($pop = 100$) and the number of generations in each epoch is 5 ($c = 5$).

We designed two algorithms to compare with the GA-lifetime. The first one is also a distributed GA whose objective is to minimize the total power consumption of the cluster. Therefore it is denoted as *GA-power*. Instead of using equation (1), the GA-power uses a fitness function as the following.

$$fitness = \begin{cases} 0 & \text{if } Prob < \mathrm{Prob}_{th} \\ 1/\sum_{i \in \Delta_k} pow_{i,k} & \text{otherwise} \end{cases}.$$

The second one is a heuristic algorithm which selects and allocates task based on the power versus detection probability ratio. For each task, it first selects the sensor node that has the highest power vs. detection probability ratio. Then it arranges the available tasks based on the descending order of this ratio. From the beginning of the list the algorithm selects the tasks one by one and assigns them to the sensor node, which is the most power efficient, until the overall detection probability of the cluster exceeds the user defined threshold $Prob_{th}$. In our experiment, the $Prob_{th}$ is set to 99.9%. The same threshold is applied to two other programs as well. We applied the above mentioned three resource management algorithms in the sensor network emulator. The lifetime of the network is recorded. The results are provided in Table 1. The first column specifies the environment setup and the last three columns specify the network lifetime (in hours) with different resource management algorithms.

ENVIRONMENT SETUP	GA-LIFETIME	GA-POWER	HEURISTIC
uniform, 8.3, 7.1, biased, 1	44752.22	41930.85	40066.12
uniform, 8.3, 7.1, biased, 2	42158.9	35224.52	35028.29
uniform, 8.3, 7.1, biased, 5	42186.89	33207.64	31848.58
uniform, 8.3, 8.3, unbiased, 1	23874.2	26462.19	26776.26
uniform, 8.3, 8.3, unbiased, 2	26828.16	31983.17	32294.36
uniform, 8.3, 8.3, unbiased, 5	27656.94	29365.77	29794.95
normal, 2, 5.5, biased, 1	480000	430909.1	450000
normal, 2, 5.5, biased, 2	511200.4	436666.7	450000
normal, 2, 5.5, biased, 5	507500	404285.7	416923.1
normal, 2, 2, unbiased, 1	14531.39	14218.28	10358.73
normal, 2, 2, unbiased, 2	13892.64	10881.77	9143.943
normal, 2, 2, unbiased, 5	15708.92	16131.24	14628.62
normal, 1, 1, unbiased, 1	2788.086	2224.231	2255.613
normal, 1, 1, unbiased, 2	2759.893	2597.652	2248.155
normal, 1, 1, unbiased, 5	2857.993	2647.201	2647.037
normal, 1.5, 4.2, biased, 1	43315.71	40533.04	34495.43
normal, 1.5, 4.2, biased, 2	49774.59	52759.19	47284.2
normal, 1.5, 4.2, biased, 5	50744.86	50134.21	48893.41

Table 1 Network lifetime under different algorithms

Figure 6 shows the percent lifetime improvement of GA-lifetime relative to the heuristic algorithm. We can see that the GA-lifetime generally works better than the heuristic algorithm. The average lifetime improvement is 14.4%. The only case for which the heuristic algorithm works better than the GA-lifetime is when the detection probability and power consumption of the tasks are distributed uniformly and the network is unbiased. This is because, in this environment setup, the detection probability and power consumption have significant variety. Therefore, there exist some task-processor pairs that are much more power efficient than others. A similar reason can be used to explain why the GA-lifetime works relatively better in the environment setup with normal distribution.

Figure 7 shows the comparison between the GA-lifetime and the GA-power. The average lifetime improvement of GA-lifetime over GA-power is 6.5%. This indicates that merely reducing the power consumption is not a good way to improve the network lifetime. If a sensor node has more remaining battery, it should be allocated with more tasks even though it is not the most power efficient node that can be used to process these tasks. In another word, to extend the network lifetime, it is more important to evenly distribute the tasks. We also observe that the GA-power outperforms the GA-lifetime when the environment setup is uniform and unbiased. This shows us that these two algorithms are complementary to each other, and they can be applied in different situations.

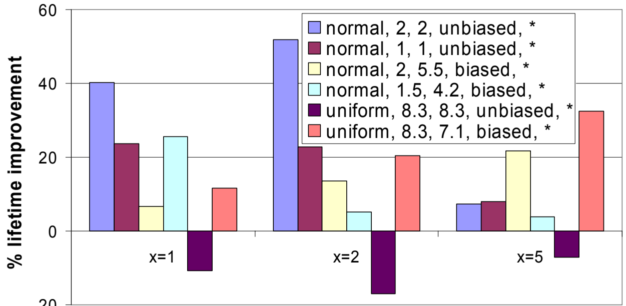


Figure 6 GA-lifetime vs. heuristic algorithm

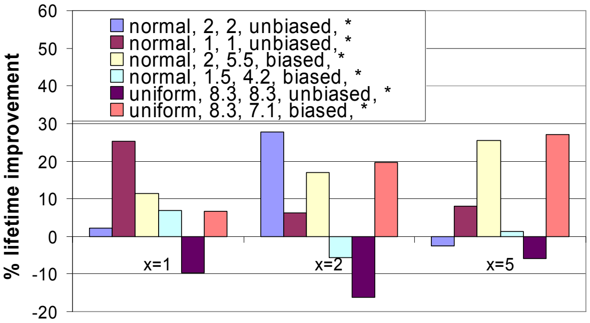


Figure 7 GA-lifetime vs. GA-power

7. CONCLUSIONS

In this paper we present a distributed GA algorithm that solves the resource management problem in a sensor network. A regression estimation model is presented that estimates the runtime of this algorithm. It is used to find the energy efficient configurations of the GA. The experimental results show that the proposed algorithm improves network lifetime by 14.4% in average.

8. REFERENCES

[1] ISTAG, "Ambient Intelligence: From Vision to Reality," Sept. 2003.

[2] I. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communications Magazine*, Volume 40, Issue 8, pp. 102-114, Aug. 2002.

[3] E. R. Post and M. Orth, "Smart Fabric, or Wearable Computing," *Proc. First Int'l Symp. Wearable Computers*, pp. 167-168, Oct. 1997.

[4] H. Feltl and G. R. Raidl, "Evolutionary computation and optimization (ECO): An improved hybrid genetic algorithm for the generalized assignment problem," *Proceedings of the 2004 ACM symposium on Applied computing*, March 2004.

[5] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energyefficient communication protocol for wireless microsensor networks," *Proceeding of International Conference on System Sciences (HICSS)*, Jan. 2000.

[6] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. Srivastava, "Topology management for sensor networks: Exploiting latency and density," *Proceeding of International Symposium on Mobile Ad Hoc Networking and Computing*, 2002.

[7] G. Mainland, D. C. Parkes, and M. Welsh, "Decentralized, Adaptive Resource Allocation for Sensor Networks," *Symposium on Networked Systems Design and Implementation*, May 2005.

[8] E. Cantu-Paz, "A Survey of Parallel Genetic Algorithms," *Calculateurs Paralleles, Reseaux et Systems Repartis*, Vol. 10, No. 2.

[9] M. Younis, K. Akkaya, and A. Kunjithapatham, "Optimization of task allocation in a cluster-based sensor network," *Proceedings of IEEE International Symposium on Computers and Communication*, 2003.

[10] http://smbus.org/.

[11] M. Pedram, "Power Minimization in IC Design: Principles and Applications," *ACM Trans. on Design Auto. of Elec. Systems*, Vol. 1, No. 1, pp. 3-56, 1996.

[12] E. Cantu-Paz, "Markov chain models of parallel genetic algorithms," *IEEE Transactions on Evolutionary Computation*, Vol. 4, Issue 3, pp 216-226, Sep. 2000.

Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis Problems

Dan Burns[1], Kevin May[1,2], Thomas Renz[1], and Virginia Ross[1]

[1] Air Force Research Laboratory
Information Technology Division
burnsd, renzt, rossv @rl.af.mil
315-330-2335, -3423, -4384

[2] Clarkson University
maykn@clarkson.edu

Abstract

This paper reports comparative results and lessons learned by developing Genetic Algorithm (GA) based optimization tools for a hard floating point (FP) and a hard integer (INT) problem, through 3 spirals on different platforms

- Software on one PC (Labview, MatLab, C) ~100X Speed-Up (FP & INT)
- Distributed software on PC cluster (C/MPI) ~30X linear Speed-Up (FP & INT)
- Hardware on FPGA (VHDL) ~500X fitness function Speed-Up (INT)

We encountered difficulties related to

- Speed-up options for double precision floating point math
- Tool maturity issues for "automated" transition from C to VHDL
- Automated synthesis of pipelined 2D systolic array for a matrix calculation
- Support for MPI or FPGA-FPGA communication on a cluster with FPGA's

Best platform depends on the type of problem. Experience highlights the potential for ~1000x speed-up of integer problem using all FPGA platforms

Outline

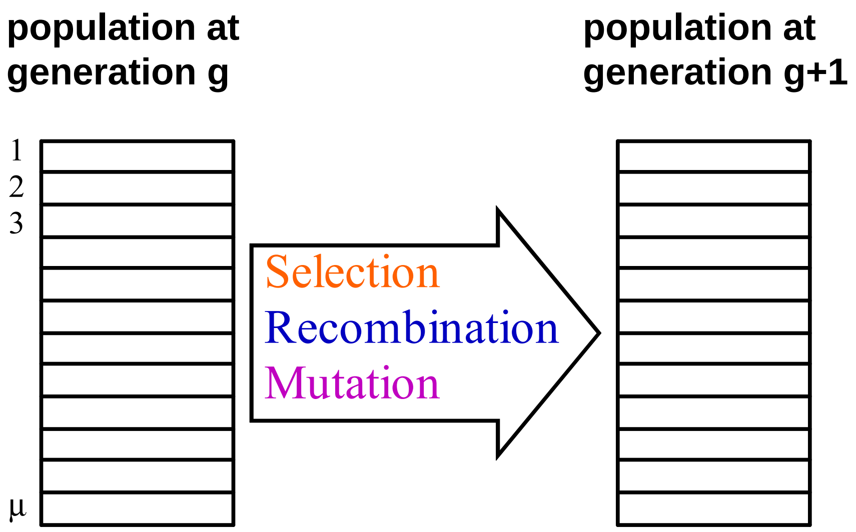
- Motivations
- Background: Genetic Algorithm and FPGA Core
- Test Case Problem 1:
  - Parameterization of Non-Linear Coupled ODE’s
  - Speed-ups: MatLab to C, PC to cluster
- Test Case Problem 2:
  - DNA Code Word Library Synthesis
  - Speed-ups: PC to cluster, PC to FPGA
- Conclusions and Future Work

Motivations

- **New/improved architectures/paradigms/engines for hard optimization problems that calculate in minutes vs. months (speed-up ~40,000x)**
- **Explore Evolutionary Computing methods for solving optimization problems, seeking extreme speed-ups over traditional approaches**
- **Good potential for original work in mission area application domains:**
  - **dynamic reconfiguration of network hosted with constrained resources**
  - **optimization of distributed database architectures and operations**
  - **assignment of routing & loading of vehicles**
  - **composing and evaluating adversary courses of action (Bayesian Belief Networks)**
  - **generation of adaptable filters for hyper-spectral imaging and compression**
  - **biological process modeling to support DARPA related to bio-hazard sensors and AFOSR/AFRL efforts and bio-molecular computing paradigms**
    - **Test Case 1: Parameterization of sets of non-linear, coupled ODE's**
    - **Test Case 2: DNA Code Word Library Synthesis for bio-molecular computing and nano-self assembly applications**
- **No turn-key commercial GA FPGA cores available**

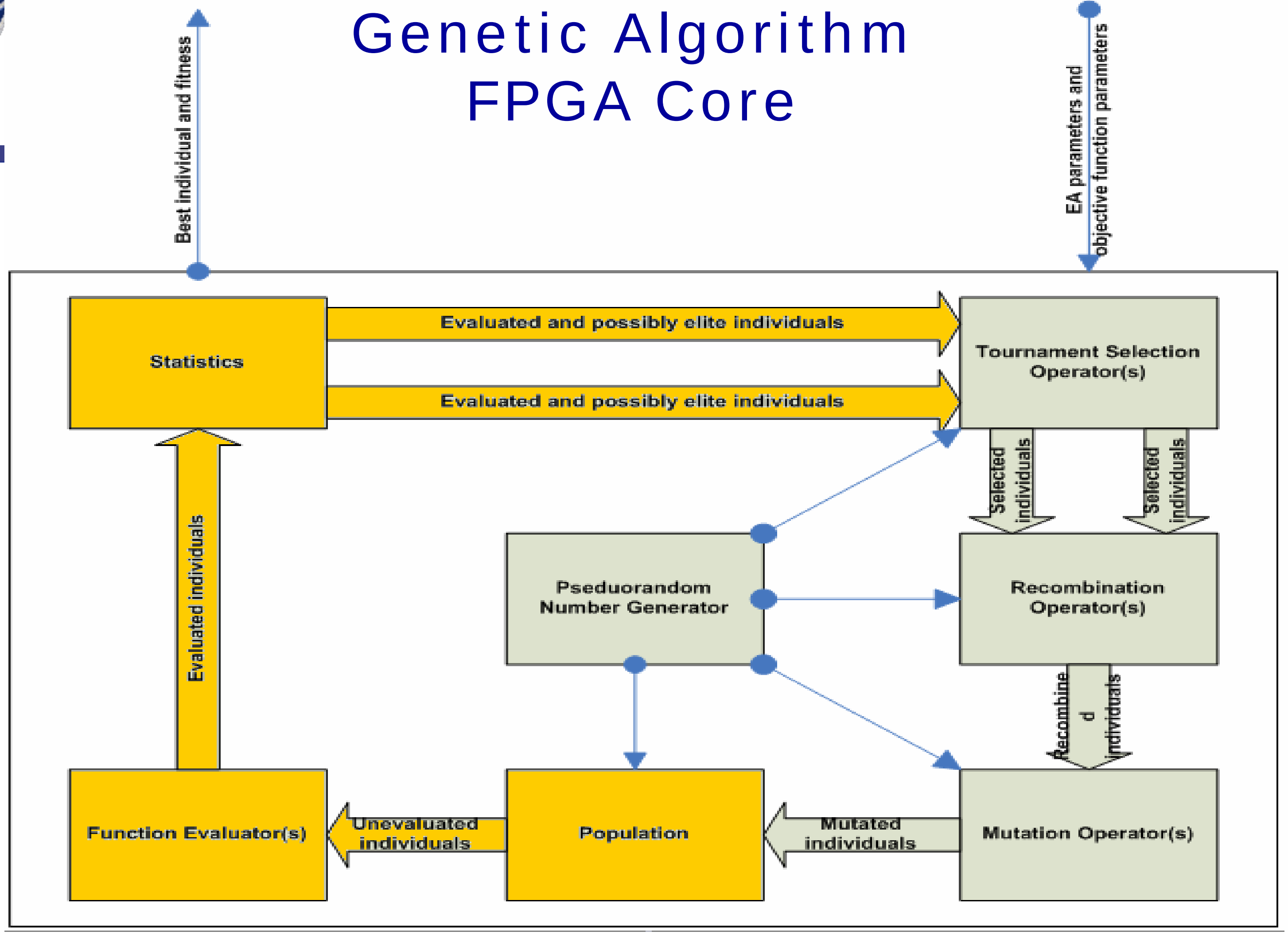
Background: Genetic Algorithm

- Inspired by processes of natural selection.
- Population initialized as collection of random individuals.
- Individuals evaluated according to fitness function.
- Genetic operators applied to population.
  - Selection: Offspring population biased toward more fit individuals.
  - Recombination: Features from multiple parents combined in offspring.
  - Mutation: Random variation added to offspring.



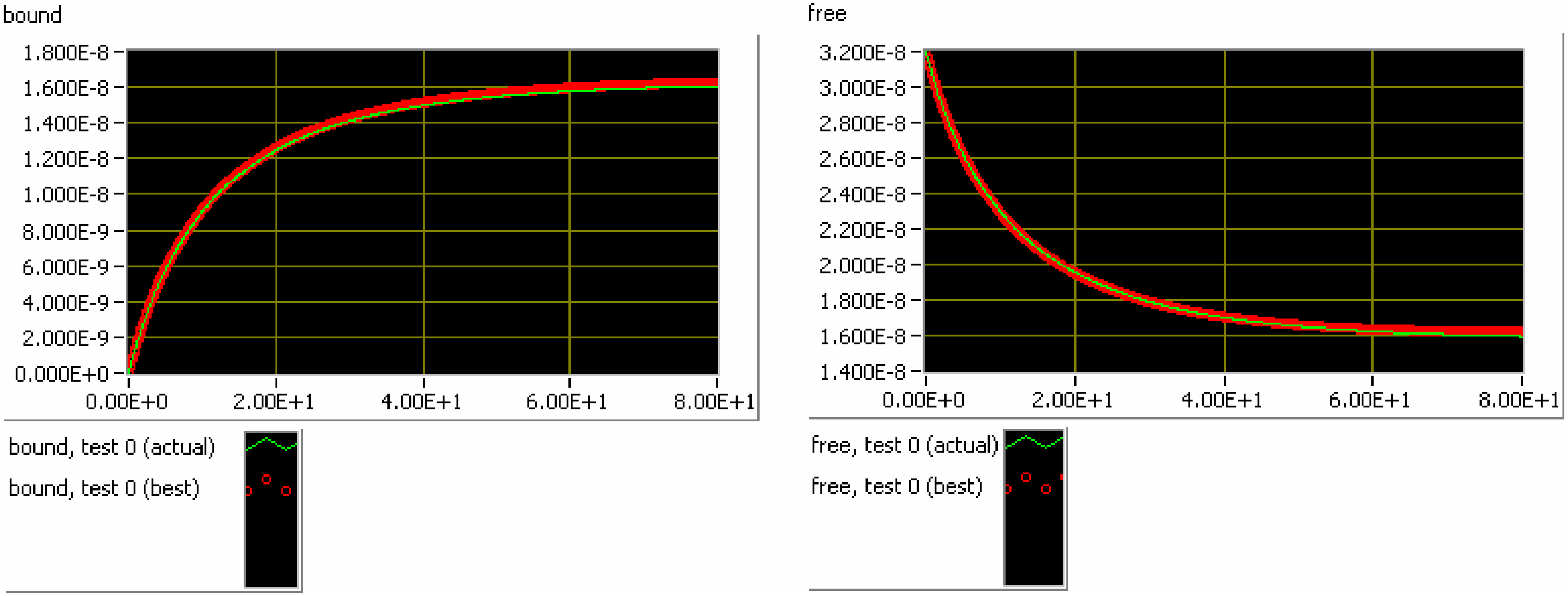
- Applied successfully as optimum-seeking techniques.
  - Useful for objective functions that are discontinuous, nonconvex, ...

Genetic Algorithm FPGA Core


Best individual and fitness
EA parameters and objective function parameters
Statistics
Evaluated and possibly elite individuals
Evaluated and possibly elite individuals
Tournament Selection Operator(s)
Selected individuals
Selected individuals
Evaluated individuals
Pseduorandom Number Generator
Recombination Operator(s)
Recombine d individuals
Function Evaluator(s)
Unevaluated individuals
Population
Mutated individuals
Mutation Operator(s)

- individuals are "records" with Chromosome , Fitness, Evaluated flag, Elite flag
- preliminary VHDL by L. Merkle/Rose Hulman, at AFRL, summer, 2004 (grey)
- completed VHDL by Kevin May/Clarkson Univ., at AFRL, summer 2005 (yellow)

Test Case Problem 1: Parameterization of Non-Linear ODE's



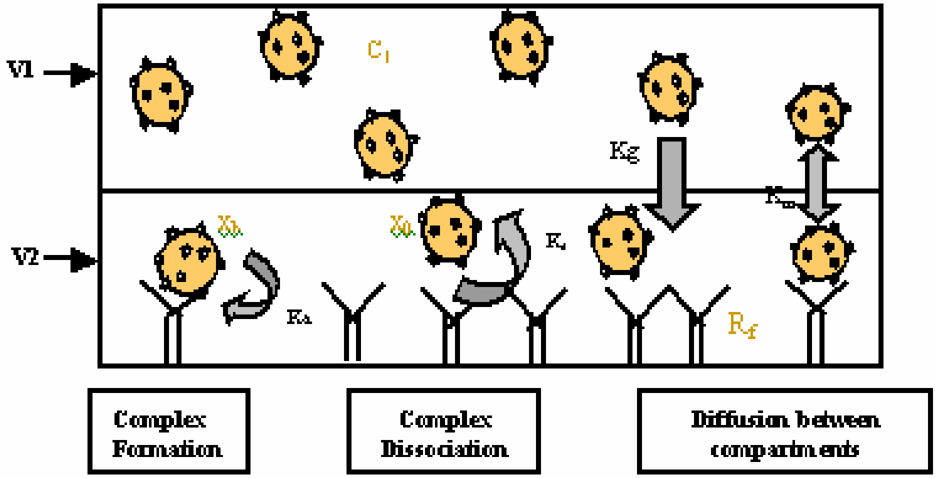
- Start with a *model*, and either measured *experimental data* or *synthetic known data* calculated using a set of parameter values for an experiment
- *Determine the constants* in the model so that the model predicts experimental or known data
- For a complex non-linear model containing many floating point parameters, this is an *NP-hard problem* that demands efficient, robust search methods.

Background: Methods for Parameterizing Non-Linear ODE Models

- Classical
  - Hill-climbing (doesn’t work if error landscape has many local minima)
  - Reduce order, do sequential partial fitting with linear tools (slow, complex)

    Rundell, A.; DeCarlo, R.; Doerschuk, P.; HogenEsch, H.; “Parameter Identification for an Autonomous 11th Order Nonlinear Model of a Physiological Process”, Proceedings of the 1998 American Control Conference, 6: 3585-3589, 1998.
  - E.g. MatLab optimization toolbox
    - Fminbnd - Golden Section search and parabolic interpolation
    - Fminsearch - Nelder-Mead simplex search method
- Evolutionary, e.g. Genetic Algorithm (GA)
  - Assigns parameters of candidate solutions to genes in individuals in a large population and *breeds better individuals* over many *generations* using *selection*, recombination or *mating* and *mutation* operators guided by a *fitness function* that grades the quality of the solutions the individuals represent.
  - Fitness function can be the least square error or maximum single point error between known data and the data calculated for an individual.
  - Floating point parameter values searched over ranges are scaled to integer gene values for the GA to manipulate.
- Both methods require many trial solutions of the model. This motivates us to pursue speed-ups using a cluster or hardware accelerator.

Analytical Two Compartment Model of Antigen/Antibody Binding at Surfaces



Eqn. for C1 $$\frac{V_1}{S}\cdot\frac{dC_1}{dt} = -(k_m(C_1 - x_0) + k_g C_1)$$

Eqn. for x0 $$\frac{V_2}{S}\cdot\frac{dx_0}{dt} = -k_a x_0 R_r \quad +k_d x_b \quad +(k_m(C_1 - x_0) + k_g C_1)$$

Eqn. for xb $$\frac{dx_b}{dt} = k_a x_0 R_r \quad -k_d x_b$$

Eqn. for Rf $$R_r = R(1 - N_a x_b D^2)$$

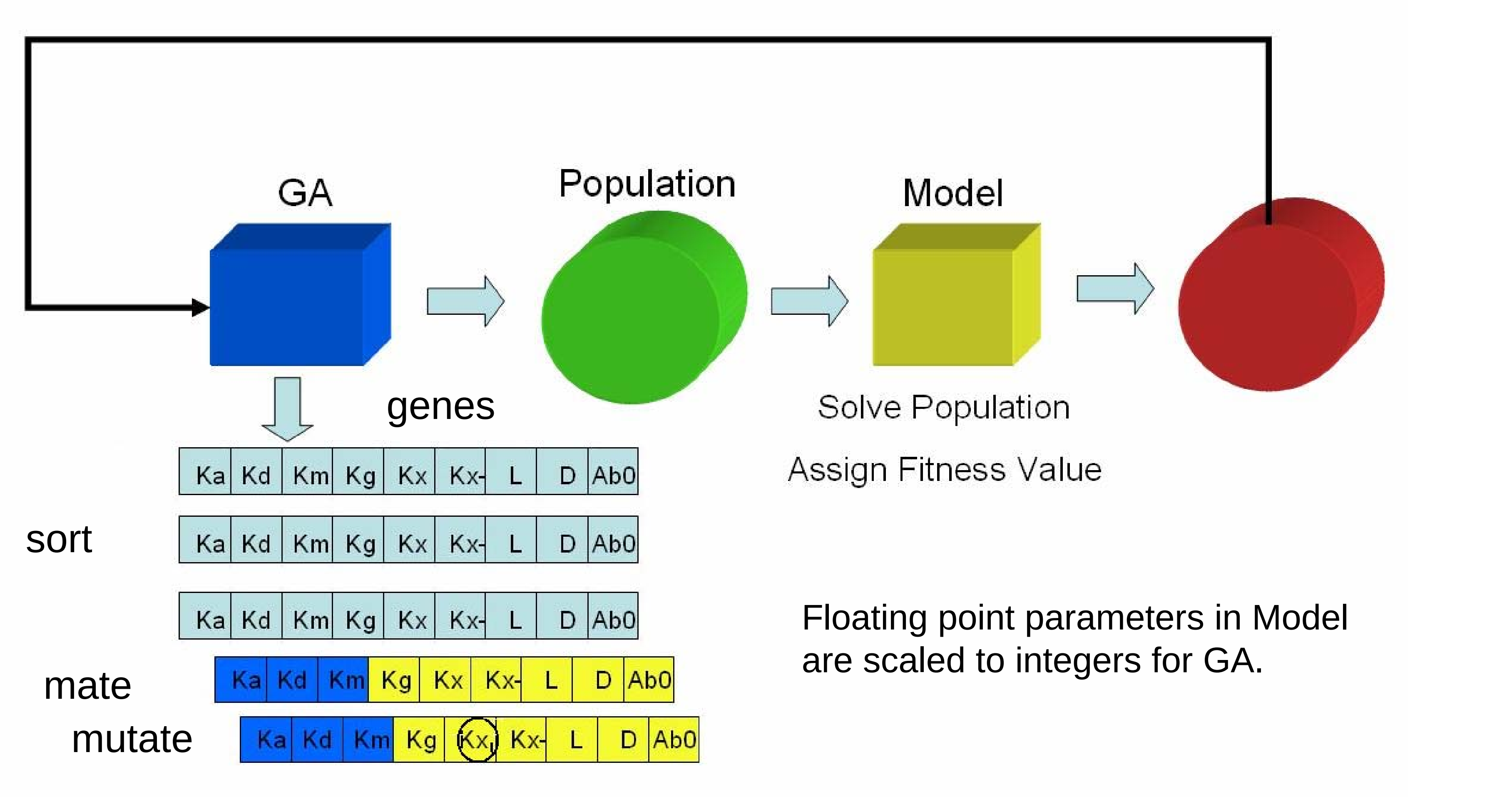
(Note: R:Initial Ab density; D: Ag diameter; Na: Avogadro's number)

Figure 5: Lumped 2-Compartmental Model

x0*xb term in **Eqn. for Rf in Eqn. for xb makes the system non-linear**

Zheng, Y.; Rundell, A., "Biosensor Immuno-surface Engineering Inspired by B-cell Membrane Bound Antibodies: Modeling and Analysis of Multivalent Antigen Capture by Immobilized Antibodies, IEEE Transactions on NanoBioscience, 2(1):14-25, 2003.

GA for ODE Parameterization Problem



GA

Population

Model

genes

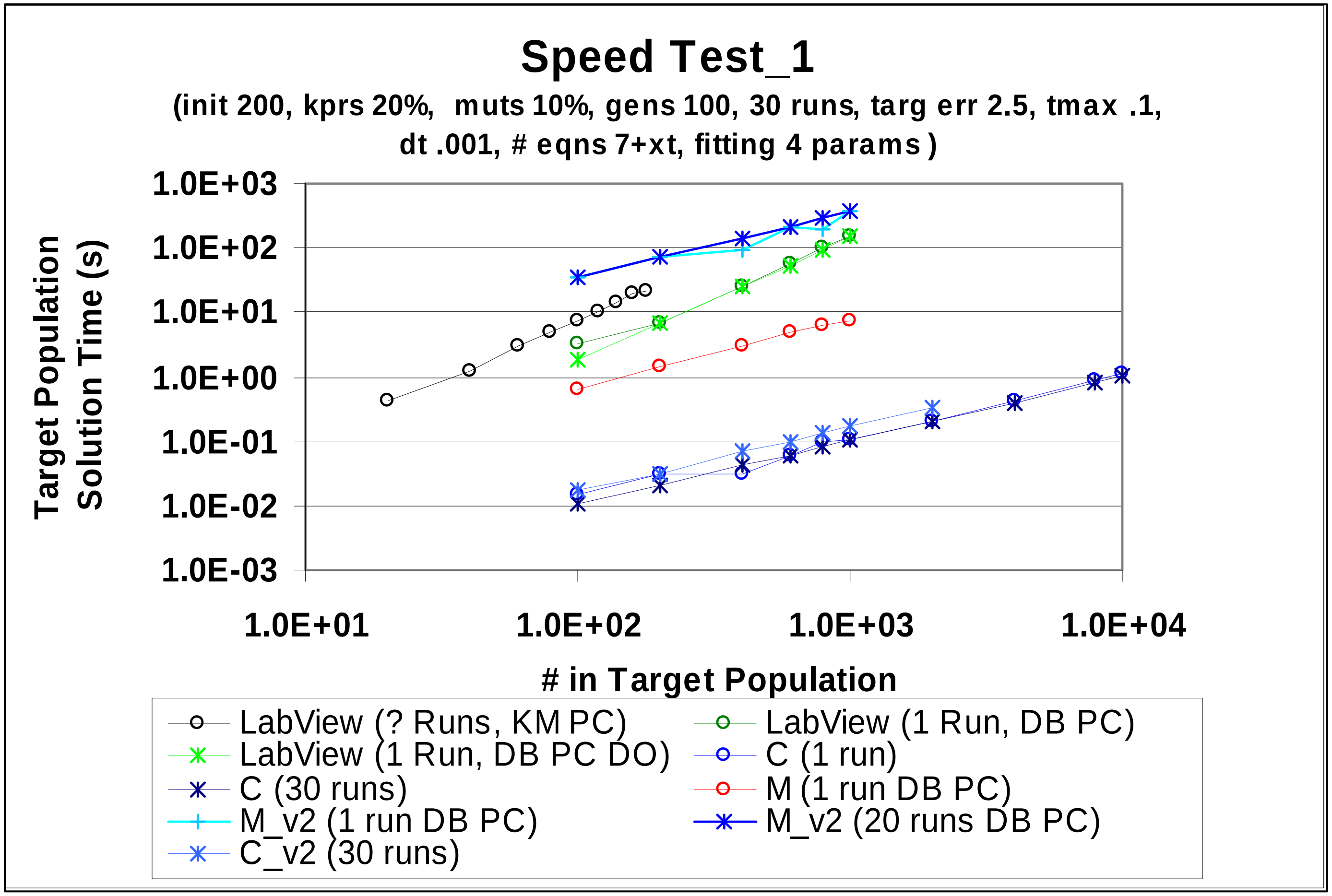
Solve Population

Assign Fitness Value

	Ka	Kd	Km	Kg	Kx	Kx-	L	D	Ab0
sort	Ka	Kd	Km	Kg	Kx	Kx-	L	D	Ab0
	Ka	Kd	Km	Kg	Kx	Kx-	L	D	Ab0
mate	Ka	Kd	Km	Kg	Kx	Kx-	L	D	Ab0
mutate	Ka	Kd	Km	Kg	Kx	Kx-	L	D	Ab0

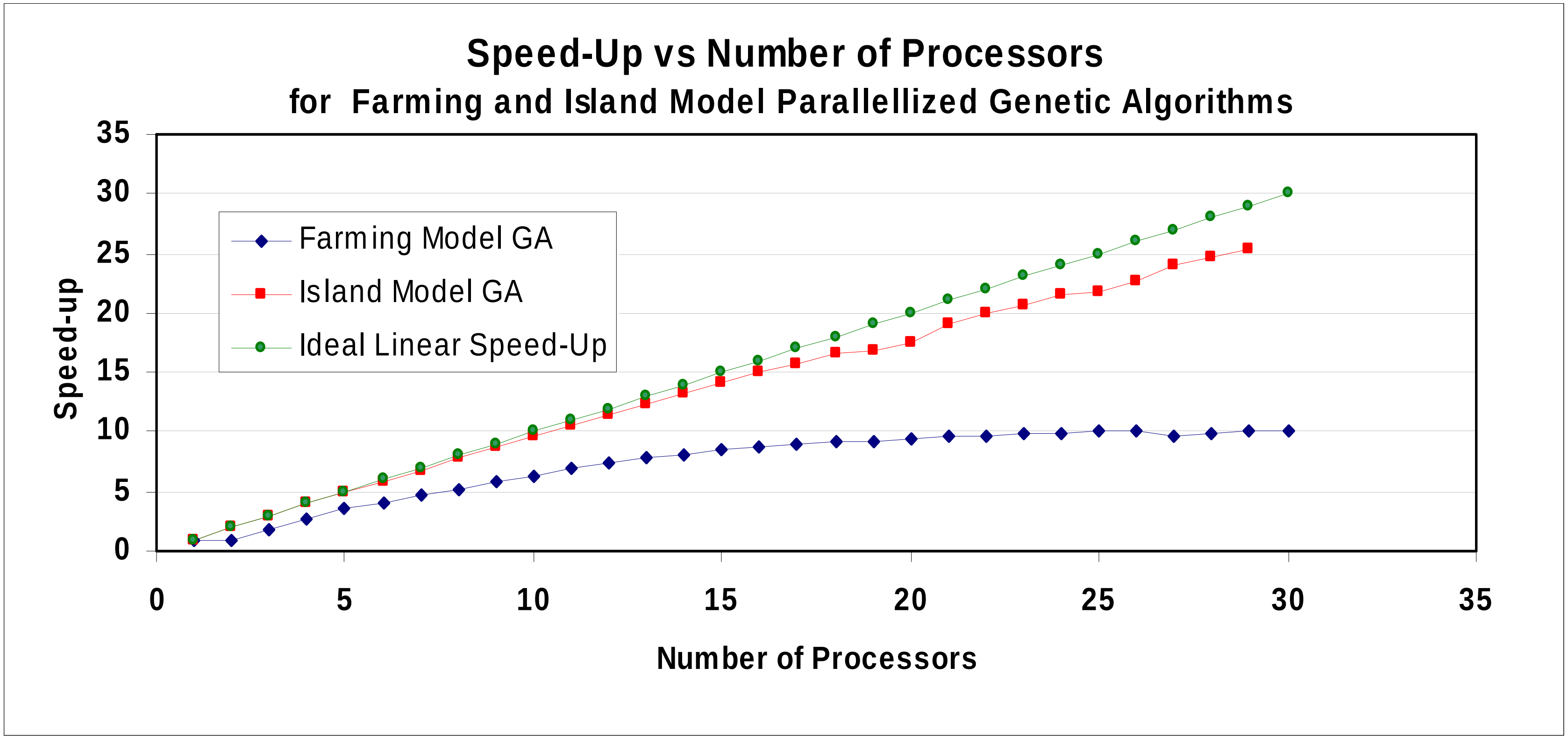
Floating point parameters in Model are scaled to integers for GA.

Speed Evaluation on PC Platform - GA Non-Linear ODE Parameterizer



Results: C ~100-1000x faster than Labview or MatLab(v6).

Speed-up Evaluation on Cluster Platform - Distributed GA Non-Linear ODE Parameterizer



- Island Model distributed GA migrates 5 best individuals around a ring topology every epoch of 40 generations
- Coded in C/MPI

GA Non-Linear ODE Parameterizer - Future Work?

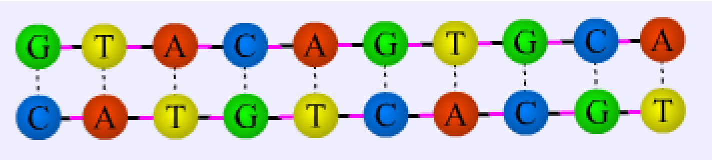
- Evaluate floating point speed-up options
- Equations involve 3-8 terms, some with 8 double precision multiplies and 1 divide
- Possible things to try:
  - Eric Cigan and Robert Anderson_"An Automated System for Floating- to Fixed-Point Conversion of High Performance of MATLAB Algorithms in FPGAs and ASICs“, MAPLD 2004, Paper 228
  - Xiaojun Wang, Miriam Leeser, Haiqian Yu,_"A Parameterized Floating-Point Library Applied to Multispectral Image Clustering“, MAPLD 2004, Paper 166
- Tune MPI and check on larger cluster

Test Case 2: DNA Code Word Library Synthesis Problem

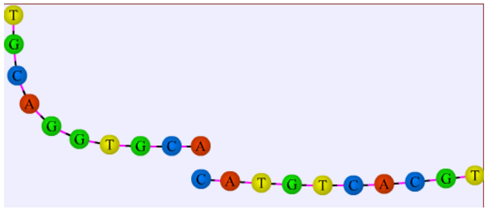
- Create library of pairs of complimentary DNA sequences which are free from undesired cross hybridization across pairs
  - e.g. A. Brenneman and A E. Condon, “Strand Design for Bio-Molecular Computers”, (Survey Paper), Theoretical Computer Science, Vol. 287:1, 2001, pages 39-58.
  - Applications in micro-array chips, schemes for computing with bio-molecules, self-assembly of nano-structures

- Great candidate problem for GA since exhaustive checking is impractical
  - Problem complexity increases as more words are desired and library constraints increase.
  - Able to keep large running population of library candidates

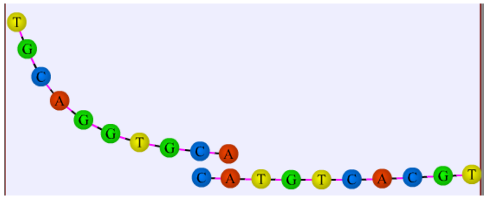
- Levenshtein Matrix is used to evaluate Candidates vs. Library and assign a fitness value to each candidate
  - Insertion – Deletion metric
  - Accounts for ~98% of total time - motivation for FPGA accelerator



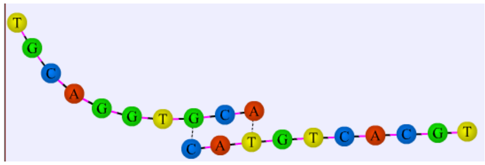
Each pair in library must bind perfectly (e.g. 10/10 bases match)



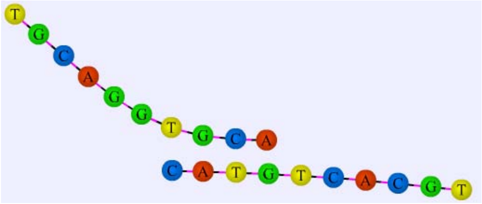
0



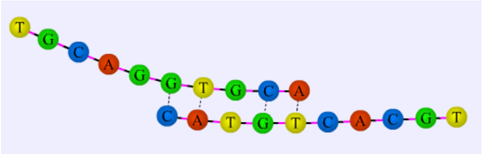
0



1



0



4

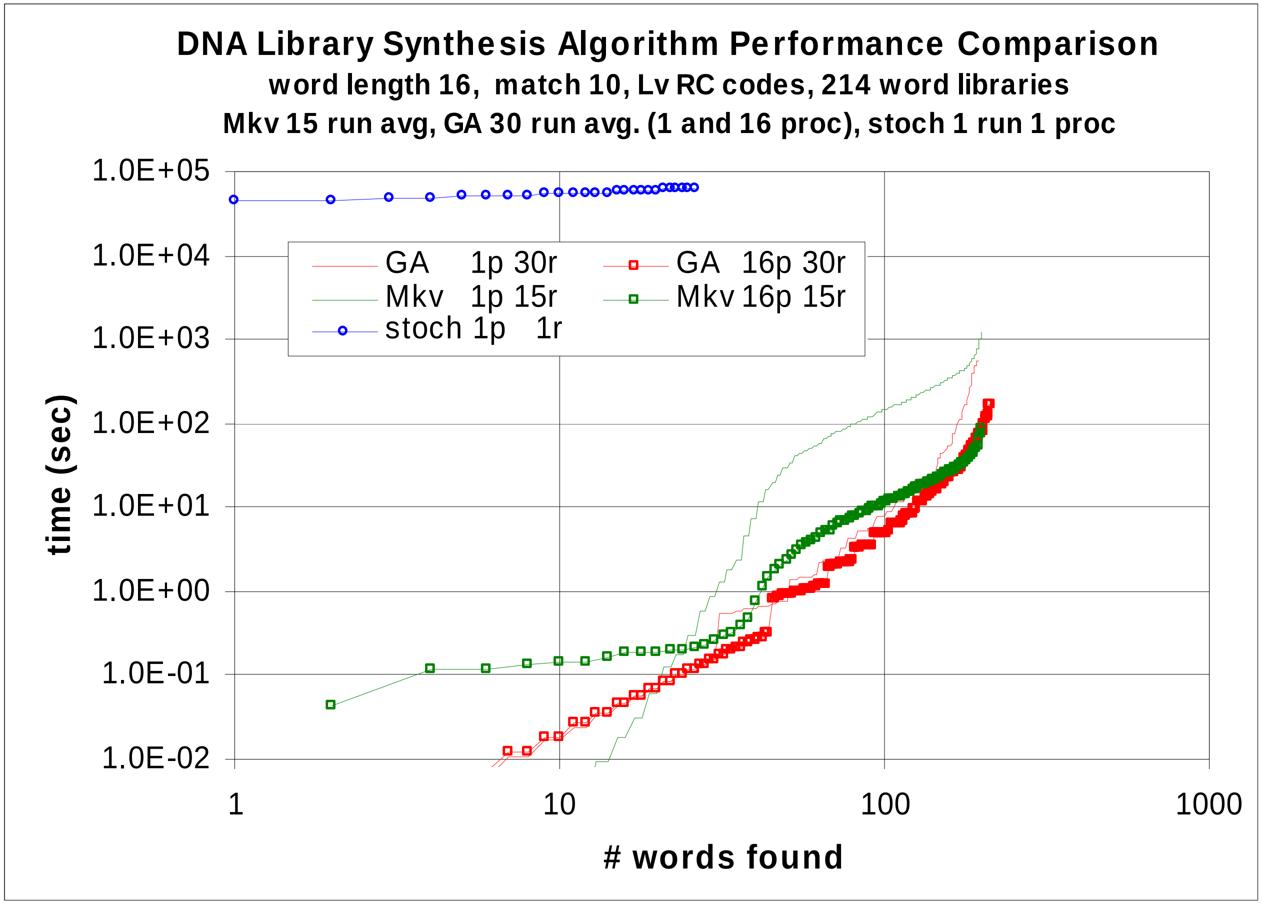
Constraint Checking

- To admit a new pair into the library, both strands in the new pair must bind poorly with all strands in all pairs already in library.
- Must check the quality of binding for all forward and reverse slidings of new stands with respect to all old strands. Quality indicated by # of binding base pairs.

Fitness Function Metrics

- Max_Match: The maximum number of complimentary bases for any sliding position, i.e. the string edit distance measured by the Levenstein Matrix.
- Number_of_Rejecters: The number of strands already in the library of strand pairs reject a new strand, using a threshold “maximum match” criteria.

Speed-up Evaluation on Cluster Platform - Markov, GA, and stochastic algorithms for DNA Code Word Library Generation Problem



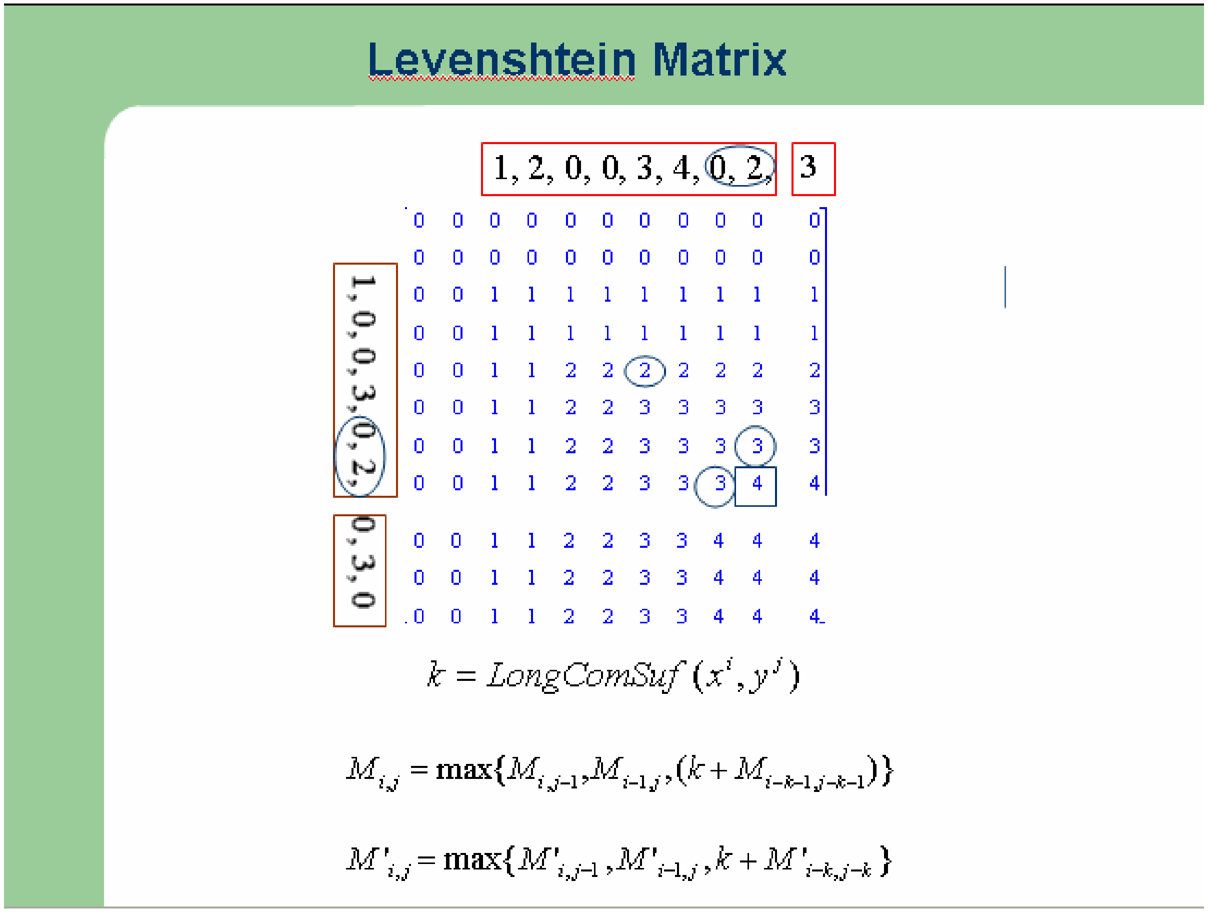
- GA (red) finds words faster than Markov for both 1 and 16 processor cases.
- Markov (green) found more words for 1 processor case.
- Stochastic (blue) is very slow due to initially full library that is improved by mutation.

Time Profiling to ID FPGA Candidate

- Distributed GA DNA Code Word Library Code

- Time profiling done with GNU gprof
- Levenstein Matrix Calculation Identified as Candidate for Hardware Acceleration

% Time	Subroutine Name
98.13	do_matrix_v6
0.65	i2s
0.44	do_checks
0.38	clean_up_pop
0.23	s2i
0.08	compliment_x_str
0.06	are_you_in_there
0.02	pop_to_word
0.00	smart_flip_2



Levenshtein Matrix

1, 2, 0, 0, 3, 4, 0, 2, 3

1, 0, 0, 3, 0, 2, 0, 3, 0

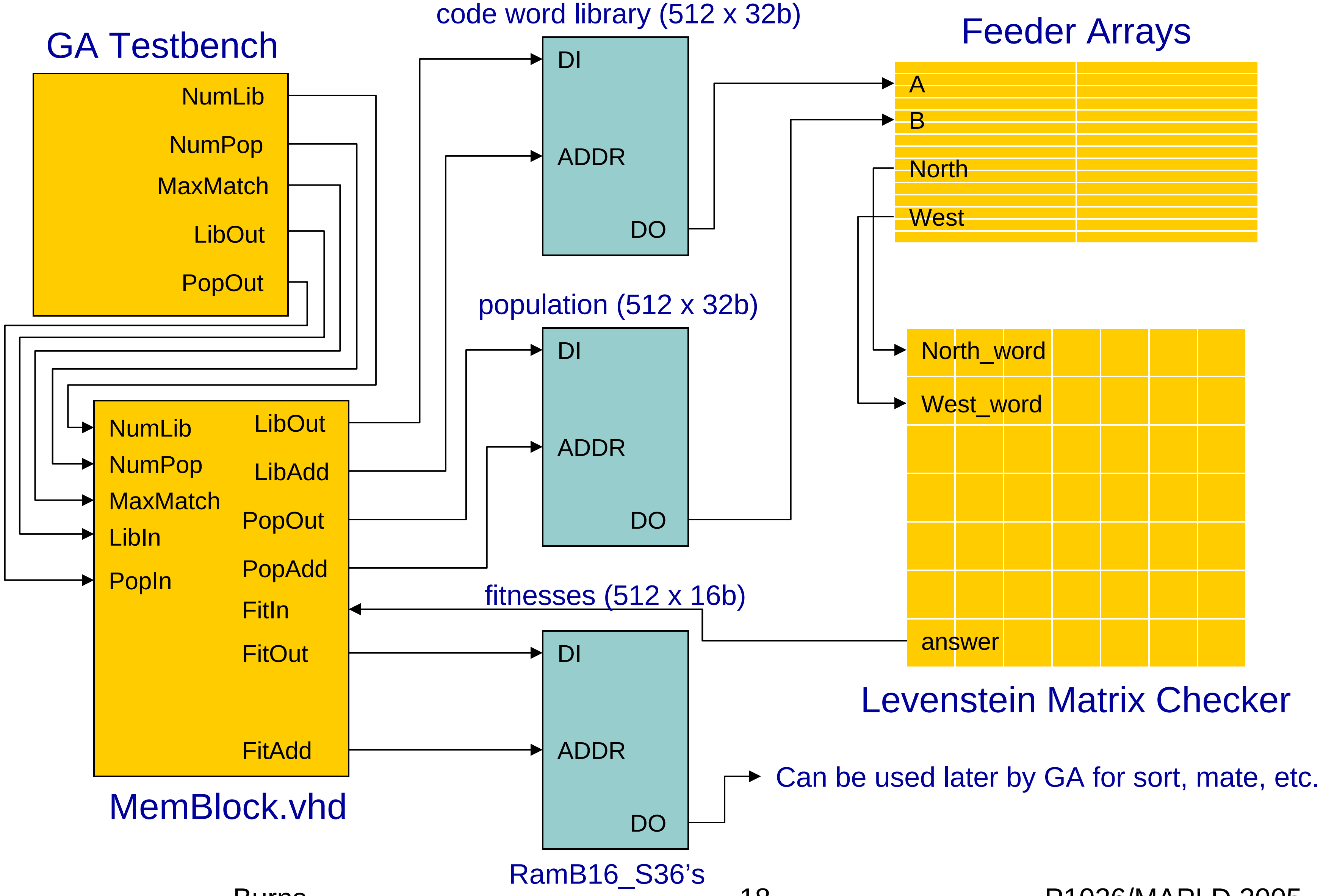
$$
\begin{array}{ccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 3 \\
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 3 \\
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & \boxed{4} & 4 \\
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 \\
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 \\
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4
\end{array}
$$

$$k = LongComSuf\,(x^i, y^j)$$

$$M_{i,j} = \max\{M_{i,j-1}, M_{i-1,j}, (k + M_{i-k-1,j-k-1})\}$$

$$M'_{i,j} = \max\{M'_{i,j-1}, M'_{i-1,j}, k + M'_{i-k,j-k}\}$$

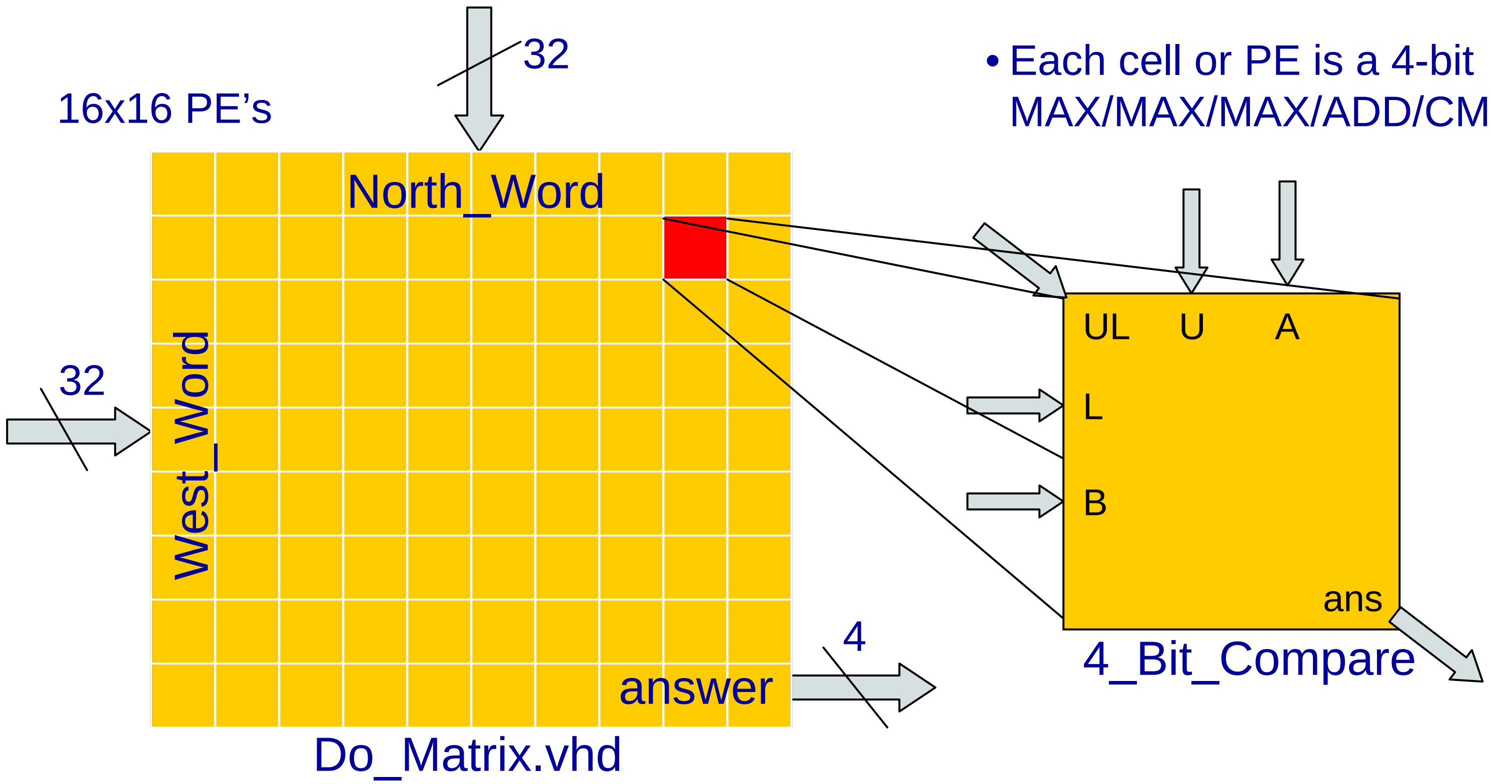
GA Core Data Path - Fitness Evaluator



Levenshtein Matrix Calculation Hardware Accelerator Versions

- Ripple Through 2D Pipelined Array
  - Present 2 words along the top and left edges of a 2D matrix of 16 x 16 Processing Elements (PE's), wait for results to ripple through matrix, read result from lower right cell, repeat for 2 new words
  - 185ns per word pair
- 2D Pipelined Systolic Array
  - Concurrent comparisons for multiple word pairs in different parts of array
  - Present 2 new words along top and left edges of 2D matrix of PE's
  - At T_n, odd half of the checkerboard loads inputs, even half calculates.
  - At T_{n+1} odd half of the checkerboard calculates, even half loads inputs
  - Input words are shifted into array down and right from edges
  - Entry of 2 bit sub-words into edges is stagger delayed by registers
  - 19.6ns per word pair, 314ns latency (16x2x9.8ns)

Levenstein Matrix Calculation Ripple Through 2D Pipeline Array



- Each cell or PE is a 4-bit MAX/MAX/MAX/ADD/CMP circuit

- **{U, L, UL} signals wired from adjacent cells' 'ans' output registers.**
- **{A, B} shift registers pass North and West Word 2bit base tokens down cols and across rows.**
- **{U, L, UL} connect to 0's depending on location on edges**

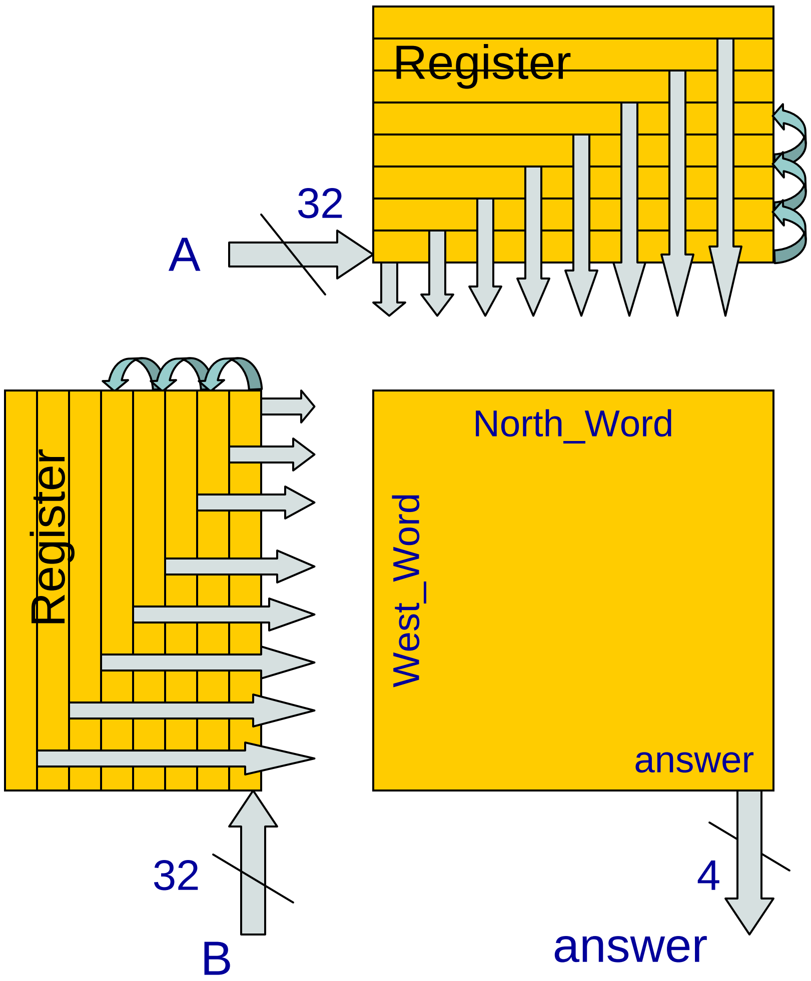
Resources and Performance:

Number of Slices:	**4273**	**out of**	**33792**	**12%**
Number of Slice Flip Flops:	**1**	**out of**	**67584**	**0%**
Number of 4 input LUTs:	**7593**	**out of**	**67584**	**11%**
Number of bonded IOBs:	**69**	**out of**	**1104**	**6%**

Estimated Delay: 185ns (5.4 MHz)

- **Much slower than 10ns target.**

Systolic Array Feeder Registers



- Feeder registers fetch pairs of North and West words from Pop and Lib memories for checking
- Calculations flow down and right along diagonals in alternating checkerboard load/calc cycles
- 2 bit base tokens are wired to edge PE {A,B} inputs in a staggered wiring pattern running up into the arrays to delay until needed by PE's
- North and West words shifted upward into word register arrays every 2 clocks, even on 1st clock, odd on next clock, in synch with alternating checkerboard input/calc in cells

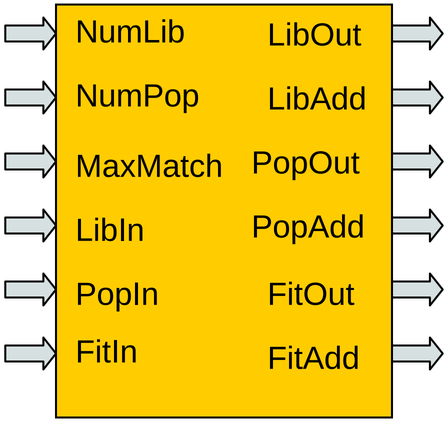
Resources and Performance:

Slices:	4150 out of	33792	12%
Slice Flip Flops:	2456 out of	67584	3%
4 input LUTs:	7300 out of	67584	10%
bonded IOBs:	69 out of	1104	6%
GCLKs:	1 out of	16	6%

Minimum period: 9.8ns (102MHz)

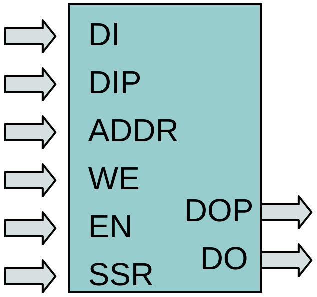
Latency is 32 clocks, answer every 2 clocks

Memory Interface



MemBlock.vhd

- The MemBlock entity is responsible for receiving the population and library from the GA and storing them, as well as the population's calculated fitness, in onboard block memories (RamB16_S36)
- The Memblock entity instantiates three of the block rams to hold the population, library, and fitnesses
- Memblock also instantiates the Feeder Arrays entity, which sequences all of the words to be compared from the population and library and returns the fitness measures.
- Each RamB16_36 can hold 18Kb (512 32bit words)



RamB16_S36*

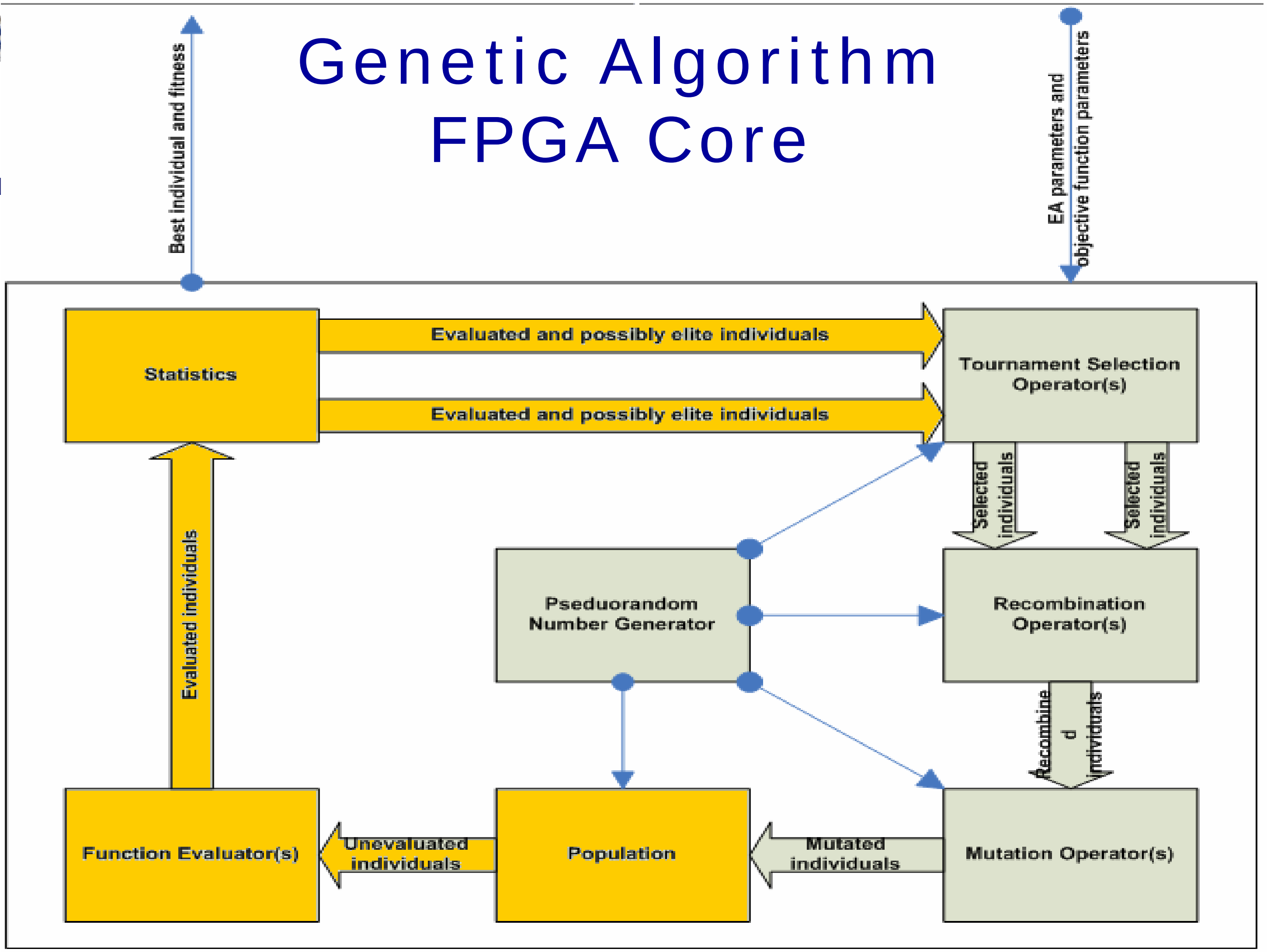
- EN (enable) and WE (write_enable) activate reading and writing and are controlled by output signals from the MemBlock entity.
- DIP and DOP parity bits and SSR synchronous set-reset pin not used here.

Resources and Performance:

Slices:	4283 of	33792	12%
Slice Flip Flops:	2544 of	67584	3%
4 input LUTs:	7532 of	67584	11%
Bonded IOBs:	98 of	1104	8%
BRAMs:	3 of	144	2%

Minimum clock period: 12.4ns (80.8 MHz)

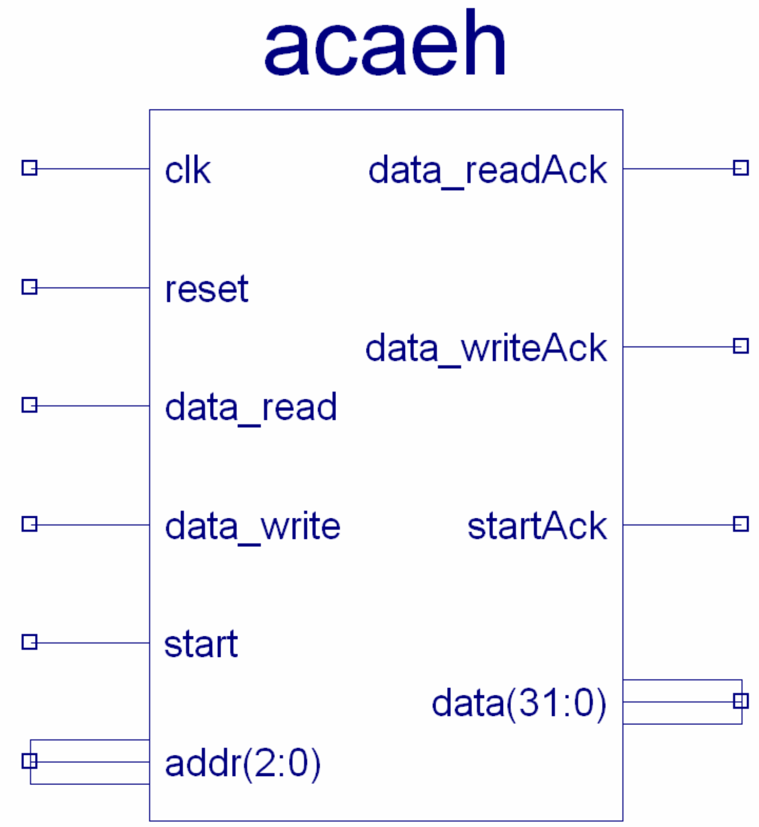
Genetic Algorithm FPGA Core


Best individual and fitness
EA parameters and objective function parameters
Statistics
Evaluated and possibly elite individuals
Evaluated and possibly elite individuals
Tournament Selection Operator(s)
Selected individuals
Selected individuals
Pseduorandom Number Generator
Recombination Operator(s)
Recombine d individuals
Evaluated individuals
Function Evaluator(s)
Unevaluated individuals
Population
Mutated individuals
Mutation Operator(s)

- Individuals are "records" with Chromosome , Fitness, Evaluated flag, Elite flag
- Preliminary VHDL by L. Merkle/Rose Hulman, at AFRL, summer, 2004 (grey)
- Completed VHDL by Kevin May/Clarkson Univ., at AFRL, summer 2005 (yellow)

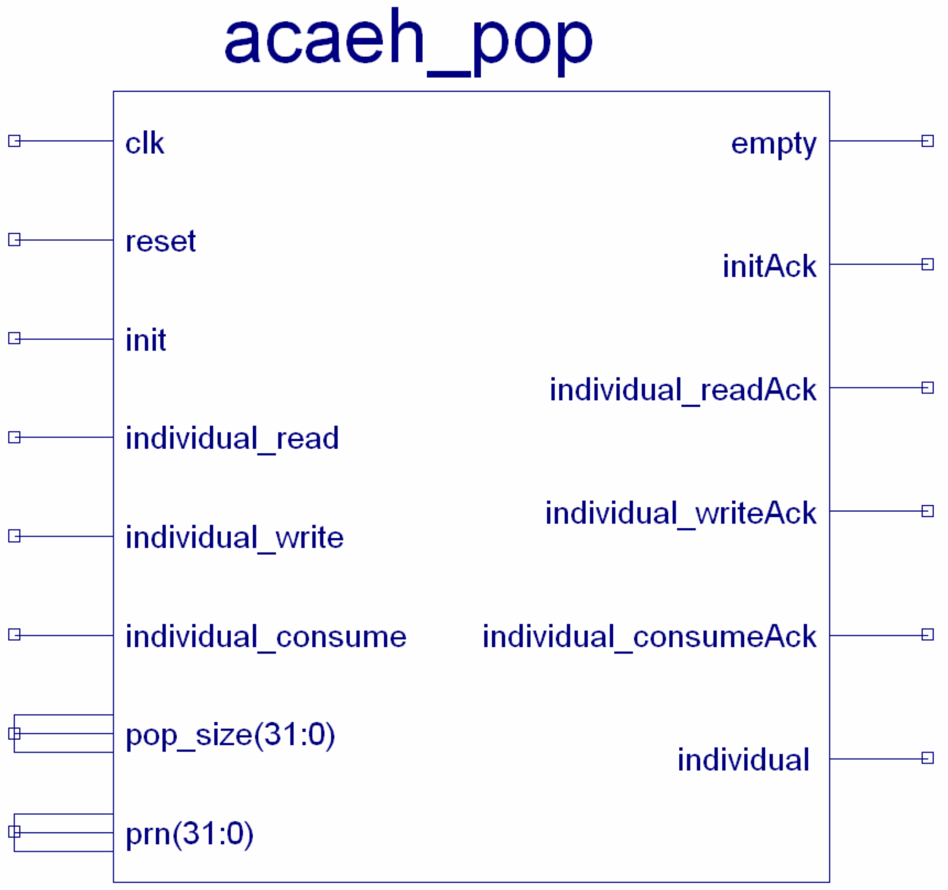
GA Core Datapath – Top-level Module

- EA parameters and objective function parameters are written into I/O ports
- When start signal is asserted, EA executes
- StartAck is asserted when EA completes
- Statistics are read from I/O ports



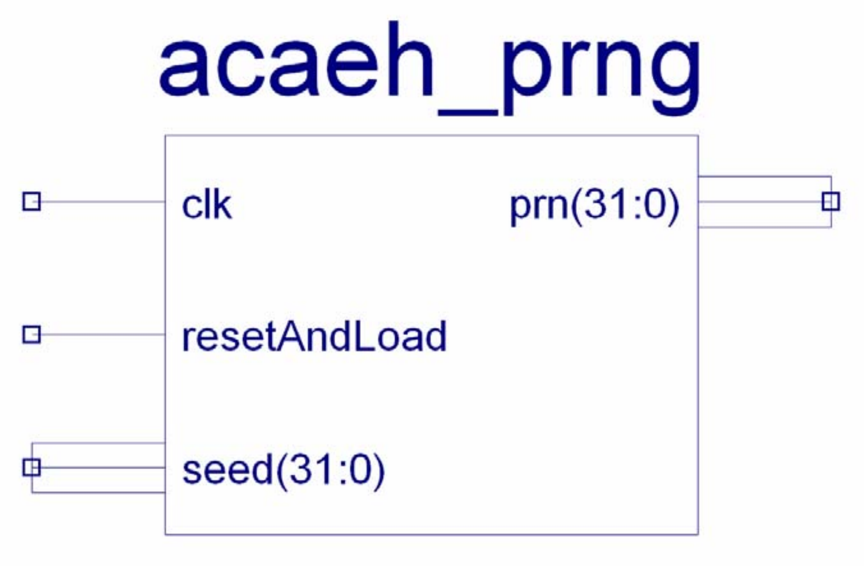
GA Core Datapath – Population Module

- Array of individuals
- Population size register
- Permutation generator
- Current permutation element register
- Current index register



GA Core Datapath – PRNG Module

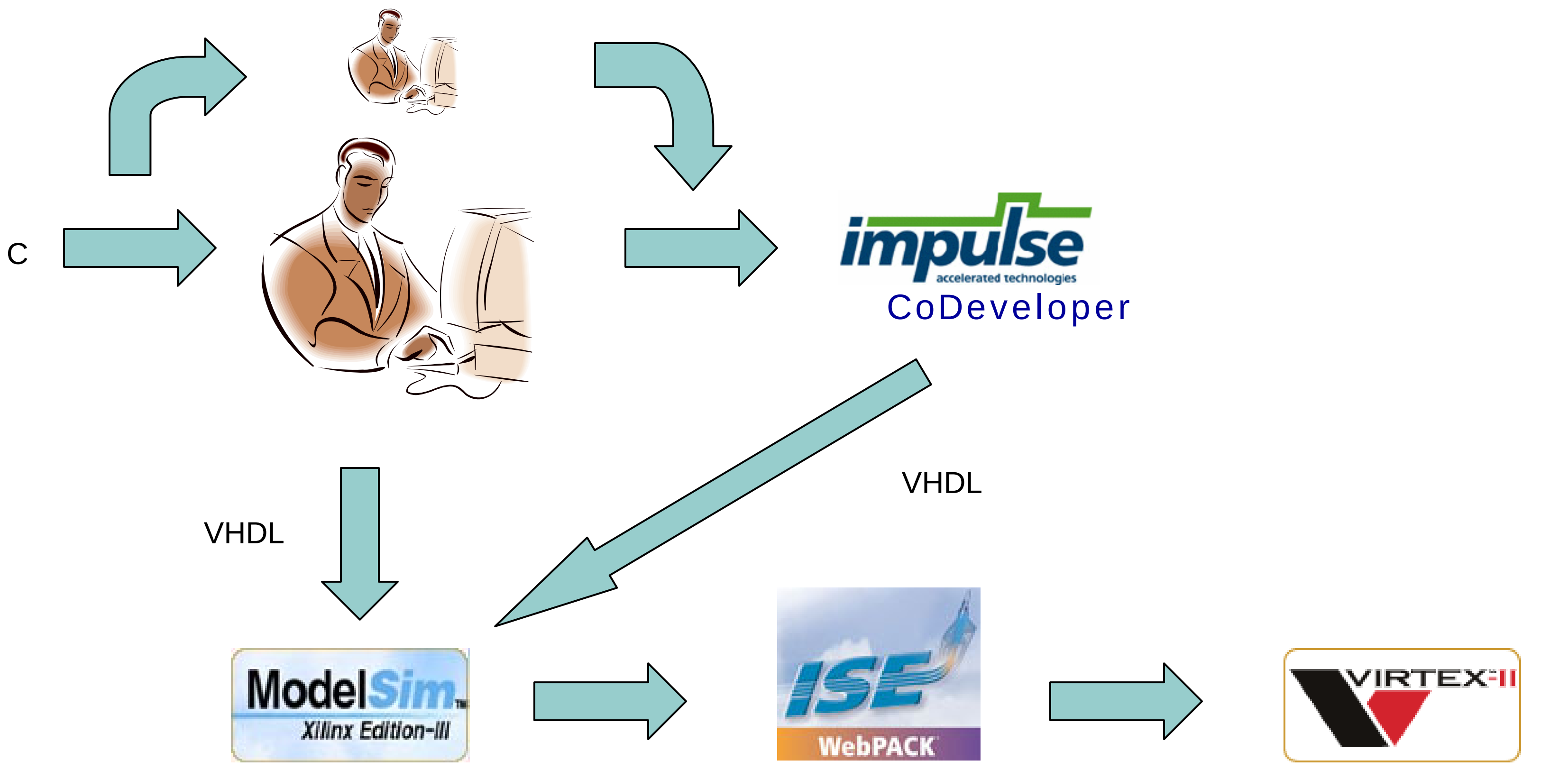
- When resetAndLoad is asserted, pseudorandom number generator is initialized with seed input
- Each clock cycle, a new pseudorandom number appears on the prn output



Speed-up and Resources for Different Platforms

	Speed-up	Resources
GA and Fitness Function (FF) on PC 0.2us GA +9.8us FF (complete)	1 **(0.2us+9.8us)**	Low
Island Model GA and FF on Cluster (complete)	30X **(30 nodes)**	High
GA on PC FF on FPGA (VHDL, synthesis complete)	50x **10us/(0.2us+9.8ns)***	Low
GA and FF on one FPGA (current work)	500-1,000x **10us/(10 to 20ns)**	Low
GA and FF on HHPC (future work)	15,000- 30,000x **(30 x 1,000x)**	High

Design Tool Paths



C

CoDeveloper

VHDL

VHDL

PROM, ACE, or JTAG

XC2V6000-4FF1517C

Collaborators

Dr. Ann Rundell, Assistant Professor
Department of Biomedical Engineering
Purdue University
West Lafayette, IN

Professor Gary B. Lamont, Ph.D.
Department of Electrical & Computer Engineering
AFIT/ENG
WRIGHT-PATTERSON AFB
DAYTON, OH

Dr. John C. Gallagher
Department of Computer Science and Engineering
Wright State University
Dayton, OH 45435-0001

Dr. Larry Merkle
Rose-Hullman Institute of Technology
Terre Haute, IN

Dr. Tony Macula
State University of New York at Geneseo
Geneseo, NY

Morgan Bishop
JEANSEE Corp.
Geneseo, NY

Conclusions

- **Speed-ups on various platforms demonstrated**
  - **Choice of programming language on PC: 100-1000X**
  - **C/MPI on Cluster: (linear)**
  - **VHDL for FPGA: 500x faster than C**

- **Distributed Island Model GA successfully applied to two test case problems**
  - **Non-Linear ODE Parameterizer**
  - **DNA Code Word Library Generation**

- **Systolic array fitness function evaluator synthesized**
  - **Levenstein Matrix Systolic Array**
  - **Clocks at 80MHz, 500x speed-up over software**

- **Modular GA Core datapath defined**

Future work

- Complete full 1 FPGA prototype
- Transition to PCMCIA, G5 platforms
- Cluster of FPGAs - communication!
- Add other EC algorithms to FPGA Core

Genetic Algorithm Hardware References

S. Scott, A. Samal, and S. Seth, “HGA: A Hardware Based Genetic Algorithm”, Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays, Monterey, CA, pp. 53-59, Feb. 1995.

- Problem: suite of 7 simple equation test case problems
- HW: Borg board using 2 XC4003’s, 8K RAM, 3 XC4005’s, 8MHz clock
- SW: Silicon Graphics 4D/440 with four MIPS R3000 CPUs at 33 MHz.
- Speed-up: 13-19x in terms of clock cycles, 100x thought to be possible.

P. Graham and B. Nelson, “Genetic algorithms in Software and in Hardware - a Performance Analysis of Workstation and Custom Computing Machine Implementations”, 1996 Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, USA, April 1996, pp. 216 – 225.

- Problem: 25 city Traveling Salesman Problem
- SW: HP PA-RISC 125 MHz workstation
- HW: 4 SPLASH 2 FPGA system with 4 memories running at 11 MHz
- Speed-up: 4X in terms of execution time, 50X in terms of clock cycles
- ARPA contract to National Semiconductor in 1994

C. Aporntewan, and P. Chongstitvatana, “A hardware implementation of the Compact Genetic Algorithm”, Proceedings of the 2001 Congress on Evolutionary Computation, 2001, Volume 1, May 2001, pp. 624 - 629 vol. 1.

- Problem: 32 bit one max problem
- SW: 200MHz Ultra Sparc II, SunOS.
- HW: Xilinx Virtex V1000FG680, 42 ns Tclk, (23.6Mhz)
- Speed-up: 1000X (0.15 sec vs 150 sec)

B.E. Wells, C. Patrick, L. Trevino, J. Weir, and J. Steincamp, “ Applying a Genetic Algorithm to Reconfigurable Hardware – a Case Study”, 2004 MAPLD, paper 169.

- Problem: 65 city Traveling Salesman Problem
- SW: 3.2 Ghz Intel Xeon processor with a large 3-level cache, Linux (Kernel 2.4.21 SMP), hosting a single user. GCC C compiler (version 3.2.2) with maximum supported level of optimization.
- HW: one Xilinx Virtex II 6000 in HC-36 Hypercomputer™ system from Star Bridge Systems, Inc. 66MHz clock.
- Speed-up: 11.4X in terms of execution time (using 8 function evaluators)

Genetic Algorithm Hardware References

M.K. Pakhira, R.K. De, "A Hardware Pipeline for Function Optimization Using Genetic Algorithms", Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, Washington, DC, June 2005, pp. 949-956.

- **Problem: suite of 7**
- **SW: classical (CGA), parallel GA (PGA)**
- **HW: pipelined GA (PLGA)**
- **Speed-up: 13-53x over CGA, 1.5-2.3x PLGA over PGA**

G.M. Megson and I.M. Bland, "The Systolic Array Genetic Algorithm, An Example of a Systolic Arrays as a Reconfigurable Design Methodology", Proceedings of the 1998 IEEE Symposium on FPGA's for Custom Computing Machines, Apr. 1998, pp. 260-261.

- **Problem: None (just doing the GA)**
- **SW: None**
- **HW: XC4036XL-09, 690 CLB's**
- **Speedup: n/a 34.3ns per gene (28.3 MHz)**

Code Word Library Design Problem References

From A. Brenneman and A E. Condon, "Strand Design for Bio-Molecular Computers" (Survey Paper), Theoretical Computer Science, Vol. 287:1, 2002, pages 39-58:

[11] R. Deaton, R. C. Murphy, M. Garzon, D. R. Franceschetti, and S. E. Stevens, Jr., "Good encodings for DNA-based solutions to combinatorial problems," Proc. DNA Based Computers II, DIMACS Workshop June 10-12, 1996, L. F. Landweber and E. B. Baum, Editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 44, 1999, pages 247- 258.

[12] M. Garzon, R. Deaton, P. Neathery, D. R. Franceschetti, and S. E. Stevens, Jr., "On the encoding problem for DNA computing," Preliminary Proc. 3rd DIMACS Workshop on DNA Based Computers, June 23-25, 1997, pages 230- 237.

[13] R. Deaton, M. Garzon, R. C. Murphy, J. A. Rose, D. R. Franceschetti, and S. E. Stevens, Jr., "Genetic search of reliable encodings for DNA-based computation," Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors), Proceedings of the First Annual Conference on Genetic Programming 1996.

Distributed and Hardware Genetic Algorithms Applied to the DNA Code Word Library Generation Problem

Daniel J. Burns
Air Force Research Laboratory/IFTC
525 Brooks Rd.
Rome, NY 13441
315-330-2335
burnsd@rl.af.mil

Morgan Bishop
JEANSEE Corp.
525 Brooks Rd.
Rome, NY 13441
315-330-1556
bishopm@rl.af.mil

ABSTRACT

Two high speed implementations of the genetic algorithm (GA) are described and their performances are evaluated on a highly constrained DNA Code Word Library Generation test case problem. The first is a distributed, or multi-deme, Island Model GA coded in C that uses the Message Passing Interface (MPI) protocol and runs on multiple processors in a cluster. The second is a single population GA coded in VHDL that implements both the GA and the fitness function evaluator in hardware on a single Field Programmable Logic Array (FPGA) chip. While the distributed GA is generally applicable to many problem types, the hardware GA is especially applicable to problems characterized by a fitness function requiring the calculation of a matrix of relatively simple integer-only or Boolean logic functions that can be efficiently implemented in a hardware systolic array.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving - Control Methods and Search - *heuristic methods*; B.2.4 [**Hardware**]: Arithmetic and logic structures – High speed Arithmetic – *algorithms;*B.7.1. [**Hardware**]: Integrated Circuits - Types and Design Styles – *Gate Arrays*

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Genetic Algorithm, distributed, parallel, hardware, systolic array, speedup, DNA Codes.

GECCO'06, July 8–12, 2004, Seattle, WA, USA.

1. INTRODUCTION

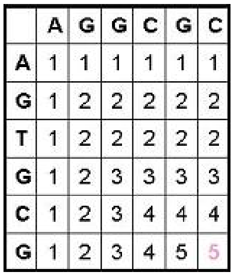
The Genetic Algorithm (GA) is one of many algorithms available attacking hard optimization problems. The simple operators used for selection, mating, and mutation suggest that the GA may ultimately hold a speed advantage over algorithms with more complex arithmetic content, especially when implemented in hardware to achieve high speed solutions. This may be important in applications where real-time decisions are critical, or where best solution times are now hours or months. Fitness function evaluation may consume a large portion of the total solution time for many problems. In such cases it may be useful to parallelize the application, or to implement it in hardware. At this point it becomes an open question whether the GA or any other algorithm can ultimately yield the fastest possible solutions. The relative advantage of one optimization algorithm over another depends in part on the set of arithmetic operations required by the algorithm, and on how efficiently the operations can be executed by a typical CPU or when implemented in special purpose hardware. For example, an algorithm requiring floating point multiplications or gradient calculations involving division may be slower than one with only integer arithmetic and Boolean operations. Similarly, the nature of the application problem fitness function also influences whether a problem is a good candidate for hardware acceleration. In this paper we describe a DNA Code Word Library generation problem that has an integer-only, array type fitness function. Then we describe two GA solvers for this problem that pursue extreme speed-ups. The first is a distributed, Island Model GA that runs on a cluster and achieves ~30x speedup. The second is a hardware implementation of both the GA and fitness function evaluator that achieves ~700X speedup. This work represents preliminary steps toward a third version that targets a hybrid cluster architecture incorporating FPGAs at each processing node. This architecture should be able to achieve speedups of over 10,000, and reduce computation times from months to minutes.

The remainder of this paper is organized as follows. Section 2 describes the test case DNA Code Word Library Generation Problem, its mapping to a GA solution, and results using a baseline software GA run on one processor. Section 3 discusses the parallel GA implementation used in the present work. Section 4 discusses the Hardware GA used in the present work. Section 5 discusses the systolic array fitness function evaluator used in the Hardware GA. Section 6 presents results on the test case problem for the two GA versions. Finally, we offer suggestions for future work in Section 7, conclusions in Section 8, and in Section 9 acknowledge others who made contributions to this work.

2. DNA CODE WORD LIBRARY GENERATION PROBLEM

DNA code word libraries contain multiple pairs of Watson-Crick complementary DNA sequences that are free from undesired cross-hybridization between any two non-complementary pairs. They play vital roles in the development of biological and hybrid information systems that operate at the nanoscale [2], e.g. in biological microarrays, nano-circuits, memory devices, robust DNA tags, in breadth-first parallel filtering schemes for solving optimization problems with bio-molecules, and in nano-fabrication schemes that would use self-assembled DNA templates to organize the layout of nano-devices. Various methods have been proposed for building such codes, including the GA [6], Markov generated [2], and Stochastic [13] methods. Recent work [9] has shown that a hybrid GA blended with Conways lexicode algorithm [4] achieves better performance than either alone in terms of generating useful codes quickly. Exhaustive checking is impractical for finding large libraries of code words of lengths greater than about 12 base pairs.

The core of difficulty for this problem is searching the very large number of candidate strands that might be added to the library, and the computational cost of calculating strand binding free energies from thermodynamics in all of the n^2 possible secondary structures which may form from any two DNA strands of equal length. The Levenshtein distance, or edit distance metric is a reasonable but computationally more efficient tool for screening candidate strings during code design. The edit distance defines the minimum number of insertions, deletions, or substitutions needed to transform one string into another. Edit distance can be considered a generalization of the Hamming distance (HD), and a minimum edit distance constraint is much more difficult to meet than HD. HD only considers substitution edits with the strands aligned fully side by side, and it can be calculated in O(2n) time. The Levenshtein distance covers all 'slidings' of two strands past each other, and it utilizes the dynamic procedure shown in Figure 1, which completes in O(n^2) time (in a sequential program).



	A	G	G	C	G	C
A	1	1	1	1	1	1
G	1	2	2	2	2	2
T	1	2	2	2	2	2
G	1	2	3	3	3	3
C	1	2	3	4	4	4
G	1	2	3	4	5	5

cell entry $M_{i,j} = \max(M_{i-1,j},\quad M_{i,j-1} , k + M_{i-1,j-1})$

where $k = 1$ if $s1_i = s2_j$ else $k = 0$, and

s1 = sequence along top row, s2 = sequence along left column

Figure 1. Calculation of the Levenshtein edit distance metric.

The DNA code word problem can be mapping to a GA by representing a strand as a string of bits using a substitution such as A=00, C=01, G=10, T=11. Thus a strand with 16 bases can be represented by a 32 bit integer. Well known GA operators such as single point crossover and bit flip mutations are applicable, with modifications. We allow single point crossover to cut only at base pair boundaries, and we use a mutation operator that selects the best of all possible single base mutations. The code design requirement specifies the desired length of the strands or words, e.g. n = 16 bases, and the desired edit distance, e.g. d = 10. In order for a pair of complimentary code words to enter the library, the new word and its reverse compliment (RC) word must meet the edit distance requirement when tested against each other and against each word and each RC word already in the library.

The GA fitness function consists of two numbers that are tabulated for each candidate word in the population. The first is the maximum of the absolute differences between the desired edit distance and the actual edit distances measured between the candidate word and its RC and every word and every RC word already in the library, which we call the max_match. The second is the number of words in already in the library that reject the new word, which we call the number_of_rejecters. A new word and its RC word can enter the library when the number of rejecters = 0. We pack the two numbers into one 32 bit word, with the number_of_rejecters in the upper bits and max_match in the lower bits to obtain a fitness metric that goes to 0 as a word gets 'better'.

Figure 2 shows the results of a typical run (on one processor node, in compiled C) in terms of # Words Found vs. Time, for good parameter tunings of both the baseline software Stochastic (top left curve), Markov (middle), and GA (lower) algorithms. Here the GA uses a population size of 100, no mating, and 1% mutations. The GA is slower at the beginning, about 10-100 times faster in the middle, and about the same near the end of the run. Since optimal (the largest possible) code word sets are not needed in practical applications, one could say that the GA is superior at finding practical code sets. A version of the stochastic method was coded and compared, but was not competitive with these times. Stochastic is similar to GA with only mutation, except that it starts with a full library and seeks to improve the fitness of all words, which requires more checks from the start.

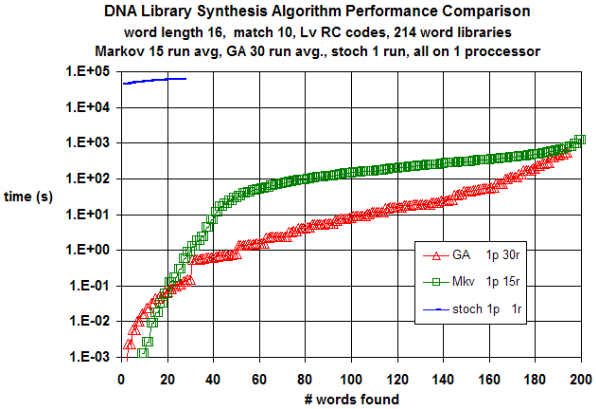


Figure 2. Relative performance of GA, Stochastic, and Markov methods.

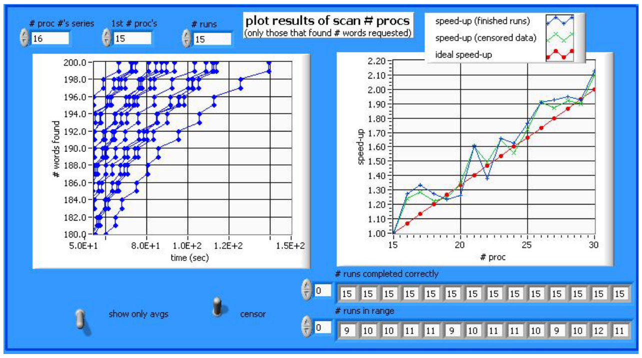
To get a sense of problem difficulty, we can calculate the number of edit distance calculations that must be made between a word and its RC and all pairs in the library during a generation. Assuming a population size of 100, a mating probability of 80%, a mutation probability 1%, and an existing library of 100 pairs, mating requires (80 children x 2 checks) x (100 words x 2 checks) or 32,000 checks. Mutation requires another (0.01 x 100) x (47 possible mutations x 2 checks) x (100 words x 2 checks) = 18800 checks. At 10 us per check this takes 0.5 seconds, but the constraints are so severe that many words must be checked to find one that can be added to the library, leading to run times of hours to assemble large libraries. Since the edit distance calculation consumes a large portion of the computation time involved in building such codes, regardless of the search algorithm used, this problem is a good candidate for speedup by parallel and hardware implementations, which are discussed next.

3. THE PARALLEL GA

Both the DNA Code Word Library generation problem and the GA are embarrassingly parallel. Much previous work has been done on parallel and distributed GAs [5,7].

We use an Island Model GA that passes the top 5 individuals to adjacent nodes in one direction in a ring configuration after epochs of 40 generations. We choose the population size so that the minimum number on individuals at each processor node was about 100. We typically use from 1 to 30 nodes in the cluster, with a total population size of 3000 individuals, split as evenly as possible among the processors. In our version of the Island Model, communication occurs between a master node and all other nodes at startup, when one of the termination criteria is met (maximum time, maximum # of generations, or desired # words found), and also in two other cases. First, when any processor finds a word, it is shared immediately with all other processors in order to keep them all working on extending identical libraries. Second, the top 5 individuals are passed around the ring in one direction at epoch boundaries. We did limited experiments with epoch lengths between 3 and 100 and found that 40 worked well. We observed that single point crossover mating was very disruptive to average population fitness, so we typically set crossover to a low value, or used only mutation. We replace clones in the population with new random individuals at the end of each generation. We used either rank or fitness based selection, with neither clearly better. Our experiments typically did 30 runs using 1 processor, 30 runs using 2 processors, etc., and we averaged the results over all 30 runs for each # of processors.

Figure 3 shows typical results in terms of the average # words found vs time (left), and a speed-up curve (right). The different plots at the left correspond to different #'s of processors, and each point in each curve is the average over 15 runs. The speedup curve shows the average time to find 200 words vs # processors used, normalized to the value for 15 processors. The 3 speedup curves show ideal linear speedup (red), uncensored speedup (blue), and censored speedup (green). Censored speedup was intended to exclude atypical runs. The 2 rows of boxes (lower right) show the number of runs at each processor value that found all 200 words (all did), and the lower row of boxes show the number of runs that finished in a time within one standard deviation of the average time of all successful runs. In this experiment the # processors was scanned from 15 to 30, the code words were 16 bases long with a desired max_match edit distance of 10, the GA used a population of 3000, there was no crossover, the mutation probability was 1% , and the 5 best individuals were passed at epochs of 40 generations. These results show an approximately linear speedup with # processors.


proc #'s series 16
1st # proc's 15
# runs 15
plot results of scan # procs
(only those that found # words requested)

runs completed correctly
0 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15

runs in range
0 | 9 | 10 | 10 | 11 | 11 | 9 | 10 | 11 | 11 | 10 | 9 | 10 | 12 | 11

Figure 3. Average # words found vs # processors and speedup curve for an experiment scanning the # processors from 15 to 30 (average of 15 runs at each # processor value).

This code was instrumented with MPI extensions that allow logging the time of the beginning and end of events at each processor during execution. This analysis is useful for optimizing the type and placement of MPI communication events. Figure 4 shows typical results after tuning the MPI communication calls. This tuning decreased the generation time from ~65 ms to ~8 ms.

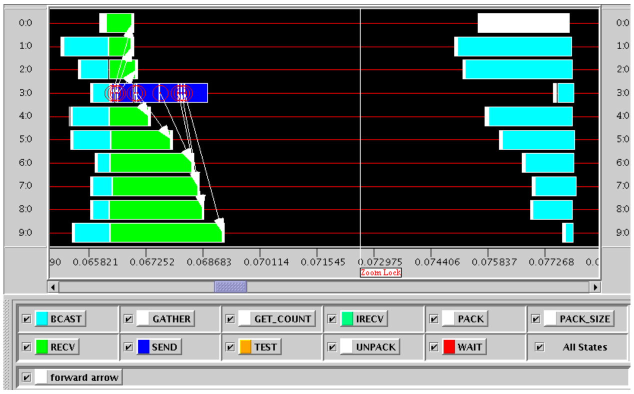

☑ BCAST ☑ GATHER ☑ GET_COUNT ☑ IRECV ☑ PACK ☑ PACK_SIZE
☑ RECV ☑ SEND ☑ TEST ☑ UNPACK ☑ WAIT ☑ All States
☑ forward arrow

Figure 4. Timing of communication (blue and green) and calculation (red line) during a GA generation for nodes 0-9. Generation boundary is at the beginning of the green area.

The communication overhead is about 25% for the cycle shown in Figure 4. There is skew and jitter in the total generation times among the processors due to startup effects, MPI response latencies, and due to the stochastic nature of the GA. This analysis is also useful for determining a population size large enough to guarantee that communication delays do not dominate the total generation time. Finally, the code was instrumented with GNU Gprof [8] to observe the duration of various tasks in the generation cycle (black in Fig. 4). This analysis showed that the subroutine that calculates the edit distance consumed 98.13% of the generation time. Therefore, the fitness function evaluator could be sped up (e.g. by hardware acceleration) by a factor of about 98.13/(100-98.13) = 112 before the GA algorithm time would be equal the fitness function evaluation time. Significant speedup beyond that would require both hardware fitness function evaluator and a hardware GA, which we discuss next.

4. THE HARDWARE GA

There has been some interest in speeding up GAs by implementing them in hardware. Reviews and examples of past work can be found in [12,11,1,14]. These studies are usually coupled with a particular type of GA and problem, and the results are often highly problem dependant. Overall speedups of 3-1000X have been reported.

Our design is different than previous work in that we have targeted a single chip FPGA implementation with the population stored and manipulated in fast, on-chip SRAMs that avoid delays associated with using off chip memory. Also, we use a systolic array on the same chip for the fitness function evaluator. We used the relatively inexpensive, commercially available Annapolis Microsystems Wildcard (PCMCIA) FPGA board that contains a Xilinx Virtex-II X2CV3000 FPGA chip. Basically, in this approach a PC executes a C 'Host' program that passes run parameters to an FPGA processing element 'PE' that implements the GA and DNA Code Word application. The Host then receives reports from the PE when words are found, and when the hardware application terminates. The PE function is described in VHDL, which we composed and simulated using the Mentor Graphics ModelSim tool. We used the Xilinx ISE Webpack or Synplicity Synplify tools for synthesis.

The FPGA design effort was focused mainly on implementing a fast function evaluator because the total execution time is dominated by fitness function evaluation. Details of the design will be described elsewhere, but here we give an outline of the main processes. They are initialization, checking fitness, picking up good words from the population into the library, mutation, decloning, and reporting results to the host. The FPGA chip contains static block RAMs (BRAMs) that can be configured as 96 separate 512 word x 32 bit RAMs. We use one BRAM to hold the population (up to 512 individuals), a second BRAM to hold the fitnesses, and third BRAM to hold the code word library. We use an overall architecture that was simply a pipeline of processes connected between sets of these 3 BRAM types. The BRAMs are dual ported, which facilitates connection between separate input and output processes. We determined that with a population size of 100 and with 100 words in the library, the time overhead for passing the entire population, fitness, and library BRAMs around the pipeline was less than 2% of generation time.

A 32 bit pseudo-random number generator (PRNG) was implemented in this design by an array of 32 32 bit linear feedback shift registers. The output word is formed by concatenating one bit from each of the 32 registers. The PRNG can be seeded by the Host to repeat a run with the same sequence of random numbers, or with a different set. A new random number is available at each PE clock edge, and all possible 32 bit numbers are represented in the sequence before it repeats. The population can be initialized with random individuals, and the library can be initialized as empty, or it can be seeded by the Host with an existing partial library.

The main GA is a tight loop of three processes, the first picking up good new words from the population into the library, the second for mutation, and the third for decloning. The pickup process looks for new good words in the population, and when it find one it moves it to the library and replaces it with a new random individual. When a new word enters the library, all of the fitnesses must be recalculated. In this step operand pairs are read from the population and library BRAMs and presented to the fitness evaluator at the PE clock rate, as described in the next Section. Each population word is checked against its own RC, and against each library word and each RC library word. The results are also analyzed and the fitness metrics are determined at the PE clock rate and stored in the fitness BRAM. During the mutation process words are selected from the population, and for each word all 47 possible single base mutations are assembled into a BRAM and their finesses are checked in a manner similar to population checking. The mutation that results in the best improved fitness is used to replace the original word, or if no mutation improves fitness, one of the 47 mutations is chosen randomly to replace the original word. When the required number of words are mutated, or if a mutation is found with 0 fitness that can enter the library, the mutation process ends and the decloning process starts. This process actually does several things. First it finds and records the best fitness in the population (without sorting). Then it replaces any words in the population that are already in the library with new random individuals. Such words will have quite good fitness, since only two words in the library will reject them, but their fitness will never be good enough. Finally, any clones in the population are replaced with new random individuals because there is no point in keeping clones given the type of mutation we use. We do not use mating, so the next generation then starts with the pickup up process.

Although we don't typically use mating for this problem, it was implemented in VHDL. In the interest of speed, we used a table look up method that implements rank based selection from the top k individuals of a population of n individuals, where k and n are BRAM addresses between 0 and 511. For example, for the case of selection from the top 10 individuals in a population of 100, a table of 10 9 bit numbers is calculated from the appropriate cumulative selection probabilities. The index of a parent is chosen by sampling a 9 bit random number from the lower bits of the 32 bit PRNG, and running an index pointing into the table up from 0 until it points to tabled value larger than the random number. One less than this index is used as the parent's index. Using this approach the Host can calculate the tabled values for any k and n <=511 and pass them in to the PE at the start of a run. This avoids resynthesis of the FPGA, which takes time.

All communication between the Host and PE is handled by an interface that is supplied as an example with the Wildcard software. It allows both the Host and PE to read and write to a common 32b register and a common BRAM that reside in the PE. Communication occurs at the beginning of a run to set GA and code design parameters, again when the PE finds each word, and finally at the end of a run. The PE records the generation each new word is found on, and also the best fitness in the population vs generation, and it passes this information back to the Host.

At this writing the entire design has been composed and simulated. The PRNG and fitness evaluator described in the next section have been synthesized. Together they use less than 20% of the FPGA chip resources, and the maximum clock frequency is higher than our 100MHz goal.

5. THE HARDWARE SYSTOLIC ARRAY FITNESS FUNCTION EVALUATOR

Systolic arrays [10] basically can perform calculations in a 3 dimensional array of cells simultaneously (2 physical dimensions and the time dimension). They are driven with fast streams of operands flowing into two edges of an array. In the case of the Levenshtein calculation, the results corresponding to any two operands flow as a diagonal front away from those edges toward the opposite corner of the array where a stream of results is read out from the lower right cell output. After a latency period of 18 clocks an answer for each set of input operands appears in the output stream at every clock period.

Figure 5 shows a block diagram of the systolic array fitness function evaluator and its feeder registers. Register arrays are needed along the top and bottom edges to sequence portions of the operands into the inputs of the edge cells at the right times. Bases in the input words at the right along the top edge and toward the bottom along the left edge must be delayed in a staggered manner before being presented to the edge cells. Each cell in the array contains the circuitry for calculating the max() function shown in Figure 1, as well as registers for passing bases in the operands down along columns and to the right along rows through the array. The array operates in a checkerboard fashion, with the cells on the even rows on even columns and odd rows on odd columns in one group, and the others in a second group. The first group loads inputs on one clock edge, and latches outputs on the next clock edge. The cells in the second group do the opposite.

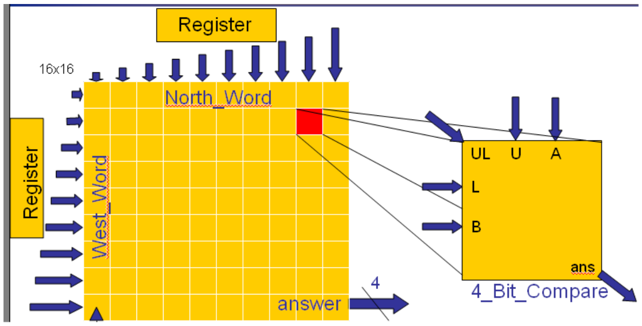


Figure 5. Block diagram of hardware systolic array for fitness function evaluation.

In the present design we actually use two instances of the fitness function evaluator, one for pickup process, and one for the mutation process. This is a side effect of writing the source code with 'hierarchical' structure (with multiple processes that can be added or debugged and changed easily), rather than in a 'flat' structure (with all functionality in one big process). Since only one hardware process can drive a signal, we need to duplicate or multiplex the inputs and outputs of any component that is used by more than one process. Multiplexing adds delay and complexity and can make routing interconnects more difficult, so we duplicated the fitness evaluator to avoid that potential problem.

6. RESULTS AND DISCUSSION

Results for the baseline software GA DNA Code Word application run on one processor were shown in Figure 2 and compared favorably with results using the best known algorithm found in the literature, the Markov method. Figure 3 also showed that the performance of the distributed version of the GA scales approximately linearly with the number of processors used. These results are shown again in Figure 6 along with a performance curve for 1 run of a simulation of the hardware version (blue), and one run of the baseline software GA run on 1 processor using the same conditions as the hardware for comparison (lower red). These two new curves were for a population size 16, vs 100 for the previous (upper) GA curve in Figure 6.

The results show that the hardware version is about 100 times faster in the early stage of the problem with few words found. The hardware version (lowest curve) was not extended because the simulation is too slow to run out to more than a few words.

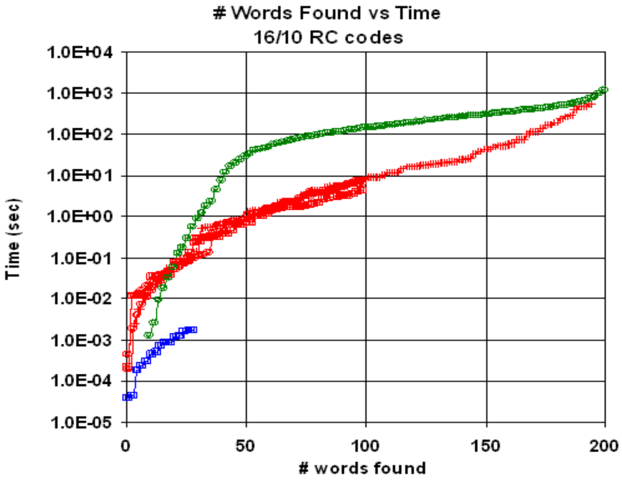


Figure 6. Relative performance of GA, Markov, and hardware GA.

To get a better idea of how the hardware version should perform in the later stages of the problem, we analyzed the simulated waveforms of the hardware version and constructed a clock cycle accurate spreadsheet model that calculates total generation time as a function of population size and the number of words in the library. We then used the model to construct a curve of generation time vs # words in the library, for the case of population size 100. The clock frequency of the FPGA was assumed to be 100MHz. We also measured the corresponding actual generation times for the baseline software GA run on one processor, also for population size 100. The results are shown in Figure 7, and they indicate that the hardware version (lower curve) should be about 700X faster than the software version (upper curve).

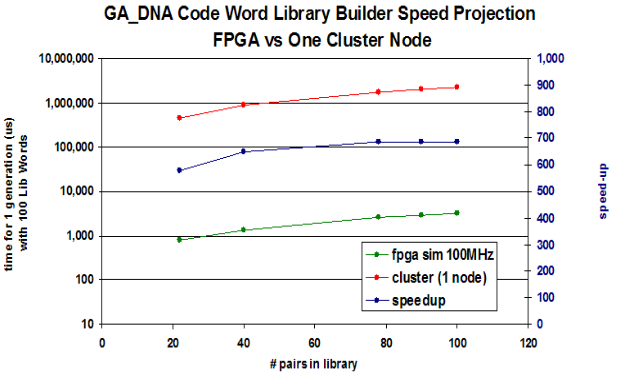


Figure 7. Comparison of generation time # words in library for software and hardware GA DNA Code Word application. (Population size = 100, composing 16/10 RC codes, no mating, 1% mutations).

7. FUTURE WORK

We plan to synthesize the hardware version and evaluate its performance. It would be desirable to add thermodynamic binding free energy calculations and other metrics used in the Markov method, such as tabulating stacked pairs, which are adjacent bases that bind between two words. This would enable a search for an improved (faster) mutation heuristic that would seek to eliminate stacked pairs. An exhaustive search option would also be useful. Presently there is no way to know whether another word exists that can be added to a library without searching. We estimate that with about 250 words in the library the present hardware systolic fitness function evaluator could check all 2^(32-1) candidate words in about 3 hours. This would actually be faster than the using the present algorithms, which can run for days before finding words. This would be useful to those interested in improving the known bounds on the size of optimal code word libraries. Finally, it would be of interest to implement a distributed hardware GA version. It might be possible to process more than one population on the same chip. Another approach would be to use fast FPGA to FPGA communication mechanisms to implement a mutli-chip distributed hardware GA. It would also be of interest to explore hardware versions of the Markov method, or other evolutionary algorithms.

8. CONCLUSIONS:

We have shown that a GA approach to solving the DNA Code Generation Problem is competitive with the best known methods in the literature. We have described a hardware systolic array implementation of the Levenstrein matrix calculation that achieves about a factor of 1000X speedup of the fitness function evaluator for this problem. We have shown that distributed and hardware GAs offer significant performance improvements of 30X and ~700X, respectively.

9. ACKNOWLEDGEMENTS

Larry Merkle of Rome Hullman Institute of Technology contributed valuable guidance and encouragement to this project during a Summer Research Faculty assignment at AFRL. Kevin May contributed to the coding of the distributed and hardware GA fitness function evaluator, and he tested the distributed GA while an undergraduate at Clarkson University studying Computer Engineering. Tony Macula of the Mathematics Dept. of SUNY Genesseo contributed suggestions for the fitness function and developed the Markov generated algorithm.

10. REFERENCES

[1] Aporntewan, C. and Chongstitvatana, P. , “A Hardware Implementation of the Compact Genetic Algorithm”, Proceedings of the 2001 Congress on Evolutionary Computation, pp. 624-629, 2001.

[2] Bishop, M. , Macula, A. , Pogozelski, W. , and Rykov, V. , “DNA Codeword Library Design”, Proc. Foundations of Nanoscience – Self Assembled Architectures and Devices, (FNANO), April 2005.

[3] Brenneman, A. and Condon, A.E., “Strand Design for Bio-Molecular Computation” , Theoretical Computer Science, Vol. 287, Issue 1, Sept. 2002, Natural computing, pp. 39-58.

[4] Brualdi, R. and Pless, V. , Greedy Codes, Journal of Combinatorial Theory,(A) 64:10-30, 1993.

[5] Cant-Paz, E. , *Efficient and Accurate Parallel Genetic Algorithms,* Kluwer Academic Publishers,Norwell, MA, 2000.

[6] Deaton, R., Garzon, M, Murphy, R.C., Rose, J. A. Franceschetti, D. R. and Stevens, S. E., Jr., "Genetic search of reliable encodings for DNA-based computation," Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors), Proceedings of the First Annual Conference on Genetic Programming 1996.

[7] De Jong, K.A. and Sarma, J. , “On Decentralizing Selection Algorithms”, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pp. 17-23. Morgan Kaufmann, July, 1995

[8] gprof: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.

[9] Houghten, S.K. , Ashlock, D. and Lennarz, J. , Bounds on Optimal Edit Metric Codes, Brock University Tech. Rept. # CS-05-07, July, 2005.

[10] Kung, S.Y. , VLSI Array Processors, Prentice-Hall, Inc., Upper Saddle River, NJ, 1987.

[11] Megson, G.M. and Bland, I.M. , “Synthesis of a Systolic Array Genetic Algorithm”, 12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98), pp. 316-320, Proceedings. IEEE Computer Society 1998.

[12] Scott, S. D. , Samal, A., and Seth, S. , "HGA: A hardware based genetic algorithm", Proc. ACM/SIGDA 3rd Int. Symp. FPGA's, 1995, pp. 53-59.

[13] Tulpan, D.C. , Hoos, H. , Condon, A. ,“Stochastic Local Search Algorithms for DNA Word Design”, Eighth International Meeting on DNA Based Computers(DNA8), June 2002.

[14] Wells, E.B., et. al., "Applying a Genetic Algorithm to Reconfigurable Hardware -- a Case Study", paper 179, 2004 MAPLD Conference, Washington, DC, 2004. http://klabs.org/mapld04/papers/e/e169_wells_p.doc

FPGA Implementation of a Genetic Algorithm and Systolic Array Levenshtein Matrix Edit Distance Calculator for Reverse Compliment DNA Code Design

Dan Burns[1], Qinru Qiu[2], Qing Wu[2], and Virginia Ross[1]

[1] Air Force Research Laboratory
Information Technology Division
burnsd, rossv @rl.af.mil
315-330-2335, -4384

[2] The State University of New York at Binghamton
qqiu, qwu @binghamton.edu

Abstract

This paper describes the design, synthesis, and performance of a single FPGA chip design that achieves a 1000X speed-up on a highly constrained DNA Code generation problem.

- **The Application:**
  - schemes for DNA assisted nano- self assembly, and information processing methods that store and manipulate data by means of biomolecules, biological assay chips.

- **The Problem:**
  - Generate optimal libraries of hundreds of minimally interacting DNA code word pairs
  - There are no known explicit construction methods for such codes
  - The bounds on optimal code sizes are unknown and experimentally determined

- **The Solution:**
  - One Xilinx XC2V3000 chip hosted on an Annapolis WildCard-II PCMCIA card on a Pentium 3 notebook platform that achieves performance on the level of a 1000 node cluster of Pentium 4s.
  - Main features include
    - a *hardware genetic algorithm* guides search over a 4.2B code word space (32 bits)
    - 32b x 32b *systolic array* calculates the Levenshtein Edit Distance between candidate and selected code words (metric in Genetic Algorithm Fitness Function)
    - All data arrays stored on-chip *all in BRAM*
    - 100MHz synthesis results using both Xilinx ISE and Synplicity Synplify tool paths
    - Hybrid algorithm using 10-20 minutes of GA followed by 1.5 hr *exhaustive search*

Abstract (cont.)

- **Problems Encountered:**
  - **Host/PE communication troublesome, fixed by double buffering registers across PCMCIA/PE clock boundaries and custom functions**
  - **Large sections of behavioral VHDL code that simulated correctly in ModelSim failed to synthesize, forcing re-write in all-clocked VHDL**
  - **Annapolis card software library supports only the Synplicity tool set**
  - **No known heuristic algorithms (including GA) are provably capable of producing globally or even locally optimal codes. Therefore, we implemented a hybrid algorithm that runs GA for a few minutes, followed by hardware Exhaustive Search (ES) that finishes in about 1.5 hours. ES uses 2 systolic array fitness evaluators to run ~260*3*4.3 Billion checks of all possible additional words, producing the only known locally optimum codes.**

- **Follow-on and Future Work:**
  - **scaling to longer word lengths, updated fitness metric calculator**
  - **a new single chip hardware multi-population distributed GA for additional speed-up**
  - **Use of the multi-pop GA chip in a cluster of FPGA's approach to yield > 500,000X speed-ups**

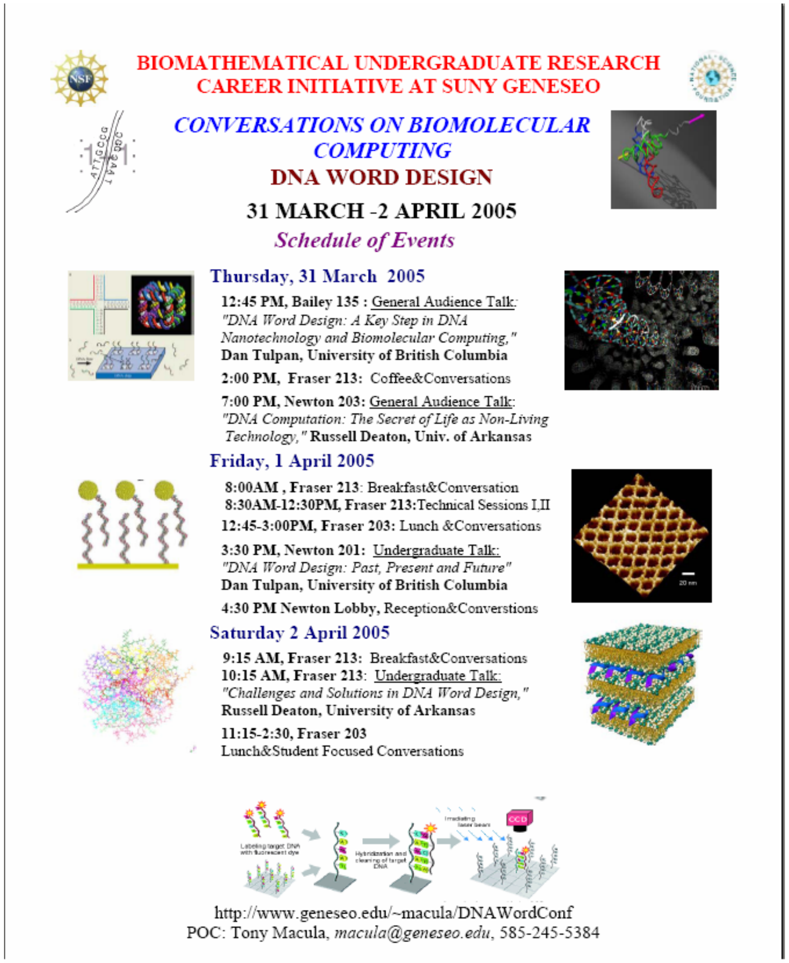
Outline

- **Motivations**
- **Background**
  - **DNA Code Word Library Generation Problem**
  - **Speed-ups: PC to cluster, PC to FPGA**
  - **Genetic Algorithm**
- **FPGA Implementation and Results**
- **Difficulties and Solutions**
- **Conclusions and Future Work**

Motivations

- **New/improved architectures/paradigms/engines for solving hard optimization problems in minutes vs. months (target speed-up ~43,200x)**
- **Explore hardware based Evolutionary Computing methods for solving hard optimization problems, seeking extreme speed-ups over traditional algorithms**
- **Pursue applications relevant to AFOSR/AFRL missions**
  - **Test Case: DNA Code Word Library Synthesis for bio-molecular computing paradigms and schemes for nano-scale self-assembly**
- **No turn-key commercial GA FPGA cores available**
- **Software distributed GA shows only linear speed-up**

Background - Uses of DNA Code Word Libraries



BIOMATHEMATICAL UNDERGRADUATE RESEARCH
CAREER INITIATIVE AT SUNY GENESEO

CONVERSATIONS ON BIOMOLECULAR COMPUTING

DNA WORD DESIGN

31 MARCH -2 APRIL 2005

Schedule of Events

Thursday, 31 March 2005

12:45 PM, Bailey 135 : General Audience Talk:
"DNA Word Design: A Key Step in DNA Nanotechnology and Biomolecular Computing,"
Dan Tulpan, University of British Columbia

2:00 PM, Fraser 213: Coffee&Conversations

7:00 PM, Newton 203: General Audience Talk:
"DNA Computation: The Secret of Life as Non-Living Technology," Russell Deaton, Univ. of Arkansas

Friday, 1 April 2005

8:00AM , Fraser 213: Breakfast&Conversation
8:30AM-12:30PM, Fraser 213:Technical Sessions I,II

12:45-3:00PM, Fraser 203: Lunch &Conversations

3:30 PM, Newton 201: Undergraduate Talk:
"DNA Word Design: Past, Present and Future"
Dan Tulpan, University of British Columbia

4:30 PM Newton Lobby, Reception&Converstions

Saturday 2 April 2005

9:15 AM, Fraser 213: Breakfast&Conversations
10:15 AM, Fraser 213: Undergraduate Talk:
"Challenges and Solutions in DNA Word Design,"
Russell Deaton, University of Arkansas

11:15-2:30, Fraser 203
Lunch&Student Focused Conversations

Labeling target DNA with fluorescent dye
Hybridization and cleaning of target DNA
Irradiating laser beam
CCD
20 nm

http://www.geneseo.edu/~macula/DNAWordConf
POC: Tony Macula, *macula@geneseo.edu*, 585-245-5384

ARKADII G. D.YACHKOV, ANTHONY J. MACULA, WENDY K. POGOZELSKI, THOMAS E. RENZ, VYACHESLAV V. RYKOV, AND DAVID C. TORNEY, "NEW INSERTION-DELETION LIKE METRICS FOR DNA HYBRIDIZATION THERMODYNAMIC MODELING", Journal of Computational Biology, Vol. 13, No. 4: pp. 866-881, May, 2006.

DNA-Based Cryptography

Ashish Gehani, Thomas LaBean and John Reif
Department of Computer Science,
Duke University, Box 90129, Durham, NC
27708-0129. E-mail: *f*geha,thl,reif*g*@cs.duke.edu

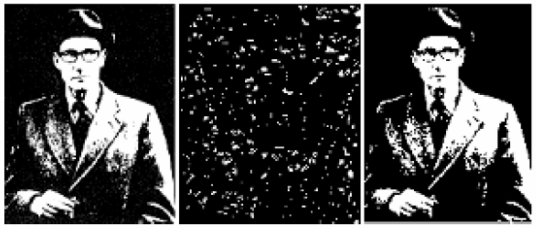
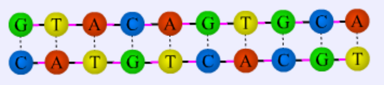
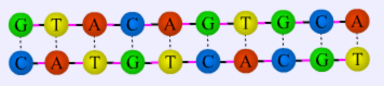
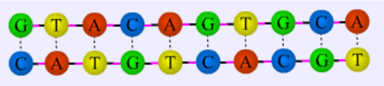


Fig. 1.4. DNA Chip Input/Output: Panel A: Message, Panel B: Encrypted Message, Panel C: Decrypted Message

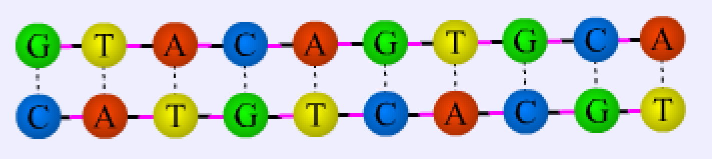
Gehani, A., LaBean, T.H., and Reif, J.H. 1999. DNA-based Cryptography. 5th DIMACS Workshop on DNA Based Computers, MIT, June, 1999.

DNA Code Word Library Generation Problem

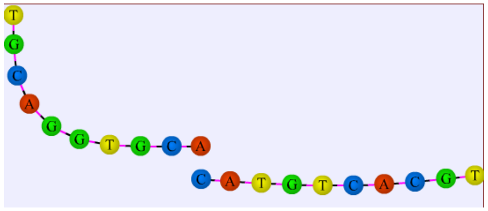
Compose libraries composed of many pairs of short DNA strands that bind perfectly within each pair, but 'poorly' across pairs or with Rev. Compliments

Library of DNA Code Word Pairs		good binding	limited binding
W1 **W1'**		**W1-W1'**	**W1-RC(W1), W1'-RC(W1'),** **W1-W2, W1-W2', W1'-W2, W1'-W2',…**
W2 **W2'**		**W2-W2'**	**W2-W3, W2-W3', W2'-W3, W2'-W3', …**
W3 **W3'**		**W3-W3'**	**W3-W4, W3-W4', W3'-W4, W3'-W4', …**

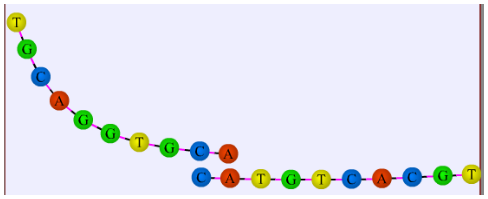
DNA Code Word Libraries are used in breadth-first parallel filtering methods for solving optimization problems with bio-molecules, in nano-fabrication schemes that would use self-assembled DNA templates to organize the layout of small components, in conceptual methods storing and accessing data in biological and hybrid information systems, in diagnostic microarrays, and as data communication codes that can correct frame registration and bit errors. (Quaternary Reverse Compliment Edit Distance Codes)



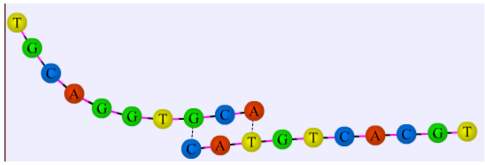
Each pair in library must bind perfectly (e.g. 10/10 bases match)



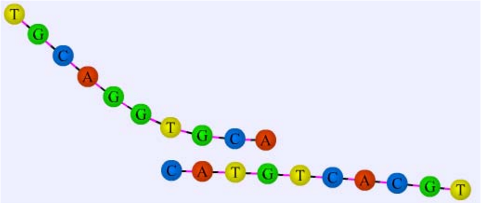
0



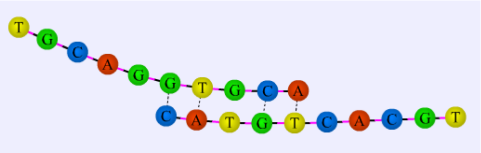
0



1



0



4

Constraint Checking

- To admit a new pair into the library, both strands in the new pair must bind poorly with all strands in all pairs already in library.
- Must check the quality of binding for all forward and reverse slidings of new stands with respect to all old strands. Quality indicated by # of binding base pairs.

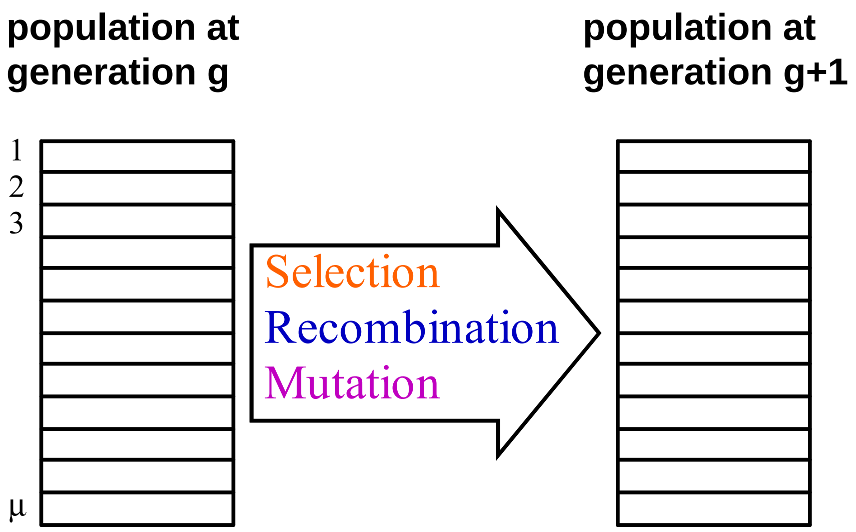
Fitness Function Metrics

- Max_Match: The maximum number of complimentary bases for any sliding position, i.e. the string edit distance measured by the Levenstein Matrix.
- Number_of_Rejecters: The number of strands already in the library of strand pairs reject a new strand, using a threshold “maximum match” criteria.

Background – the Genetic Algorithm

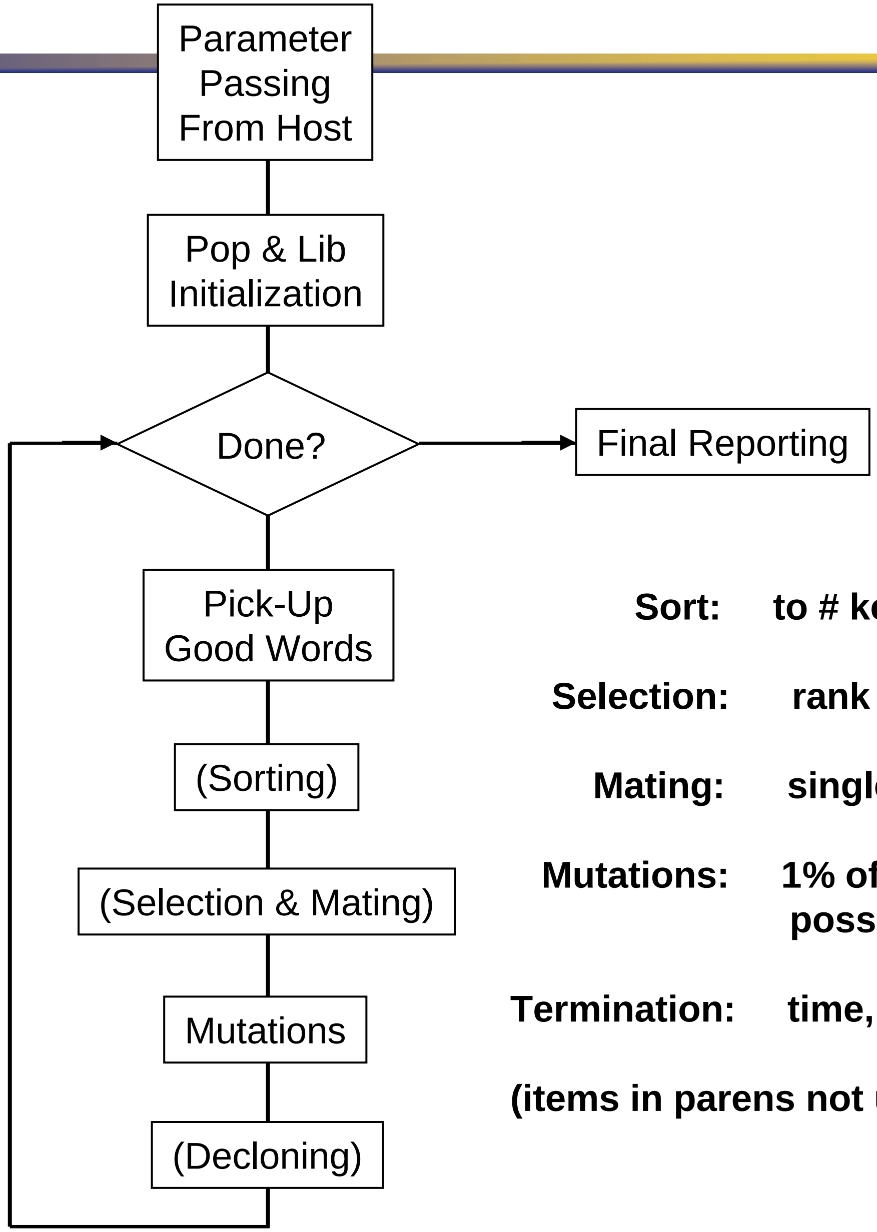
- an embarasingly parallel search algorithm that scales

- Inspired by processes of natural selection.
- Population initialized as collection of random individuals.
- Individuals evaluated according to fitness function.
- Genetic operators applied to population.
  - Selection: Offspring population biased toward more fit individuals.
  - Recombination: Features from multiple parents combined in offspring.
  - Mutation: Random variation added to offspring.



- Applied successfully as optimum-seeking techniques.
  - Useful for objective functions that are discontinuous, nonconvex, multi-objective, ...

GA/DNA Main Process Flow Chart



Parameter Passing From Host

Pop & Lib Initialization

Done?

Final Reporting

Pick-Up Good Words

(Sorting)

(Selection & Mating)

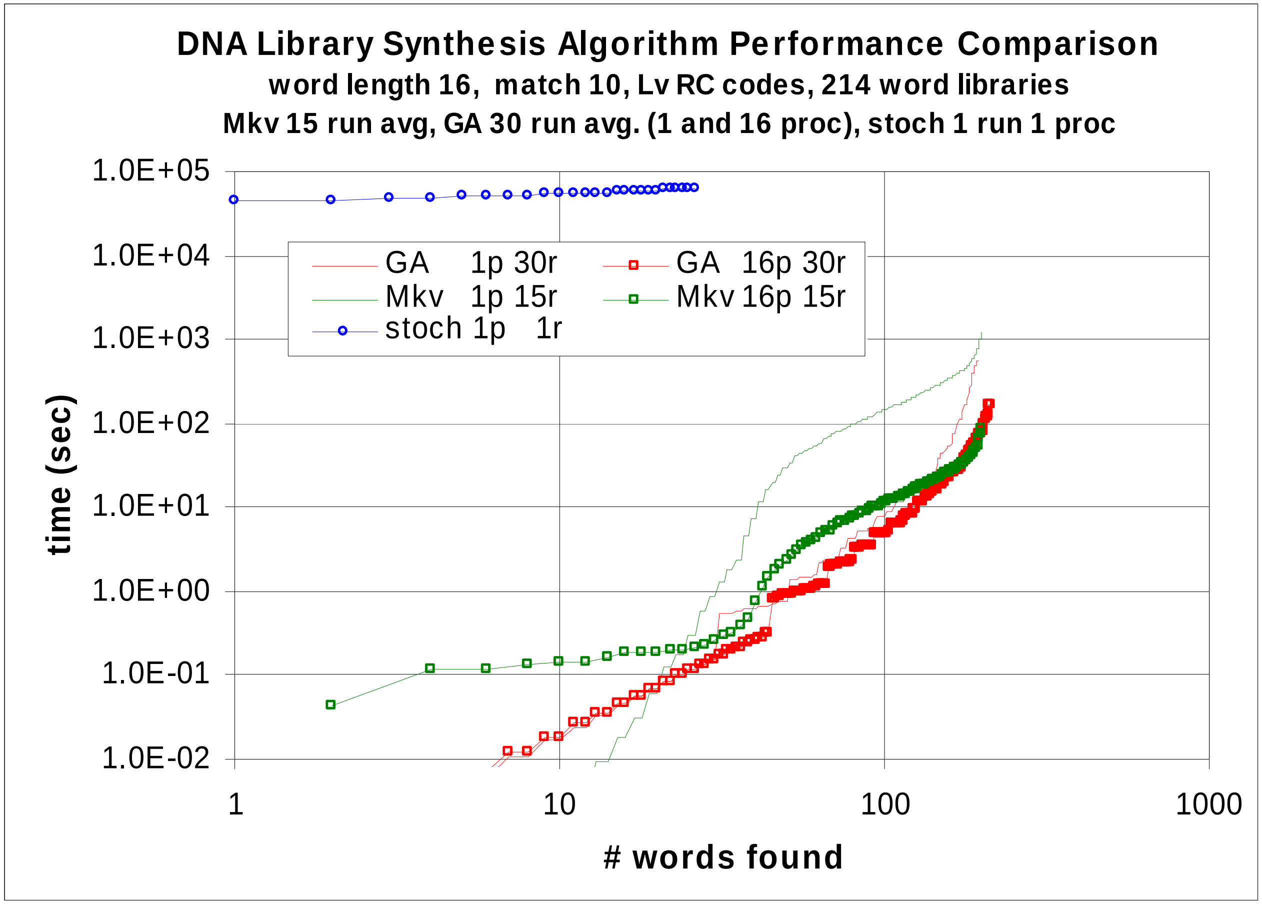
Mutations

(Decloning)

Sort:	**to # keepers from # pop**
Selection:	**rank based probability**
Mating:	**single point crossover**
Mutations:	**1% of pop indvs, best of all 47 possible single base changes**
Termination:	**time, # generations, # words**

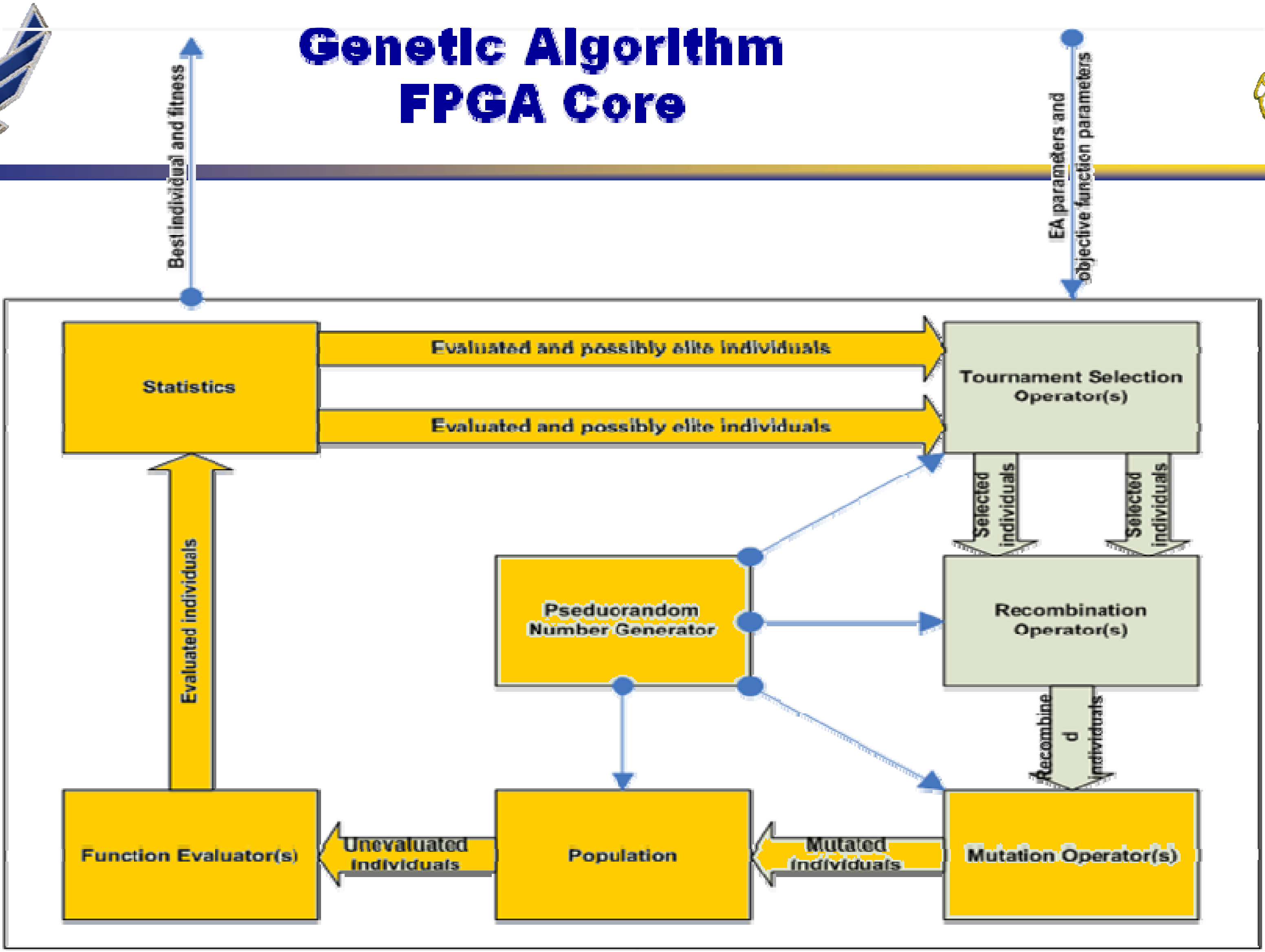
(items in parens not used in FPGA version)

Speed-up Evaluation on Cluster Platform
- Markov, GA, and stochastic algorithms for DNA Code Word Library Generation Problem



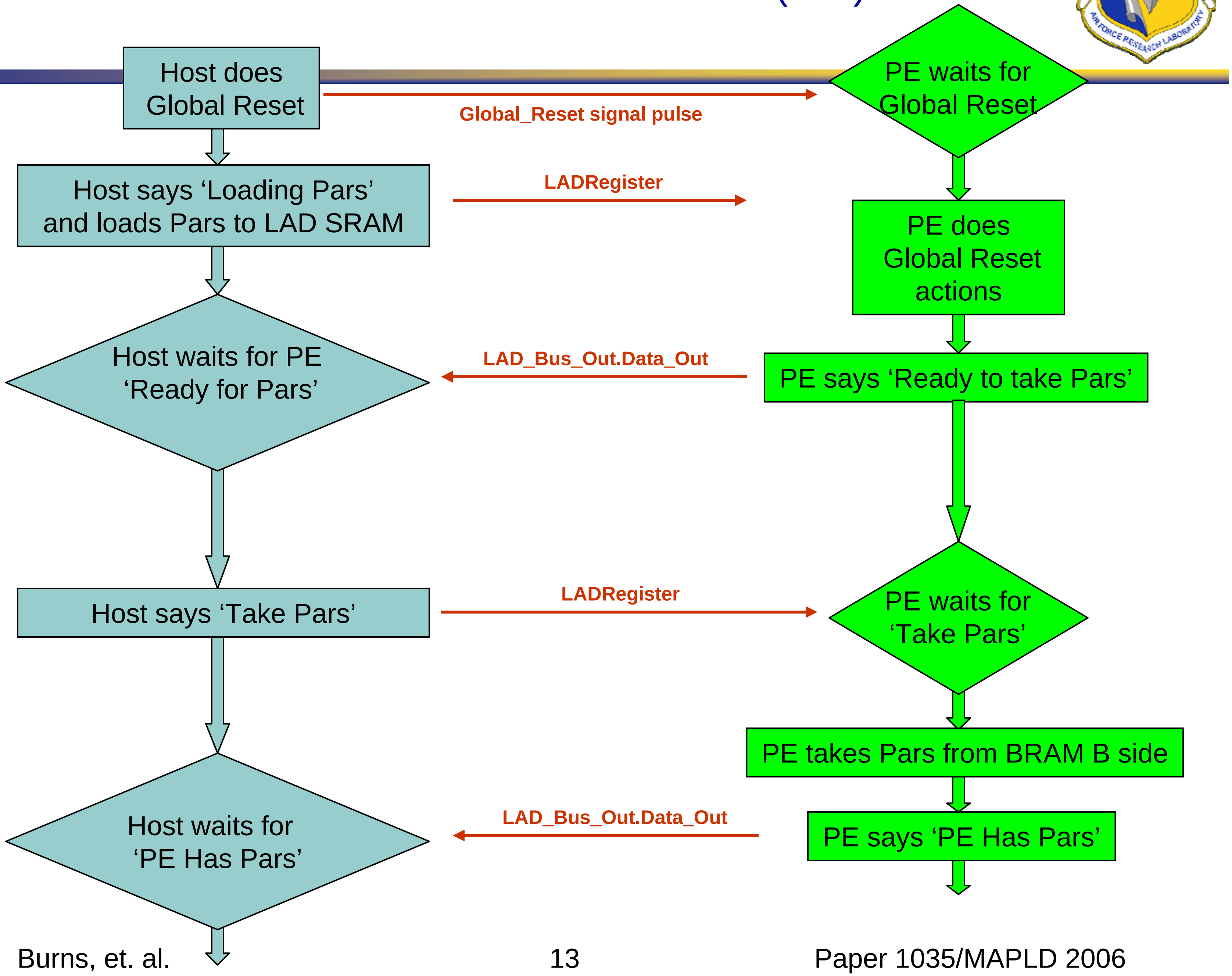
- GA (red) finds words faster than Markov for both 1 and 16 processor cases.
- Markov (green) found more words for 1 processor case.
- Stochastic (blue) is very slow due to initially full library that is improved by mutation.

Genetic Algorithm FPGA Core


Best individual and fitness
EA parameters and objective function parameters
Statistics
Evaluated and possibly elite individuals
Evaluated and possibly elite individuals
Tournament Selection Operator(s)
Selected individuals
Selected individuals
Evaluated individuals
Pseduorandom Number Generator
Recombination Operator(s)
Recombine d Individuals
Function Evaluator(s)
Unevaluated Individuals
Population
Mutated Individuals
Mutation Operator(s)

- Population Individuals are 32 bit integers that represent DNA strands
- 32 bit PRNG needed to initialize population, replace picked-up words, for selection, recombination, and mutation heuristics. Done with feedback LSR.
- Full GA shown above, but selection and recombination not used to date in FPGA.

Host/PE Process Interaction (1/4)



Host does Global Reset

Global_Reset signal pulse

PE waits for Global Reset

Host says 'Loading Pars' and loads Pars to LAD SRAM

LADRegister

PE does Global Reset actions

Host waits for PE 'Ready for Pars'

LAD_Bus_Out.Data_Out

PE says 'Ready to take Pars'

Host says 'Take Pars'

LADRegister

PE waits for 'Take Pars'

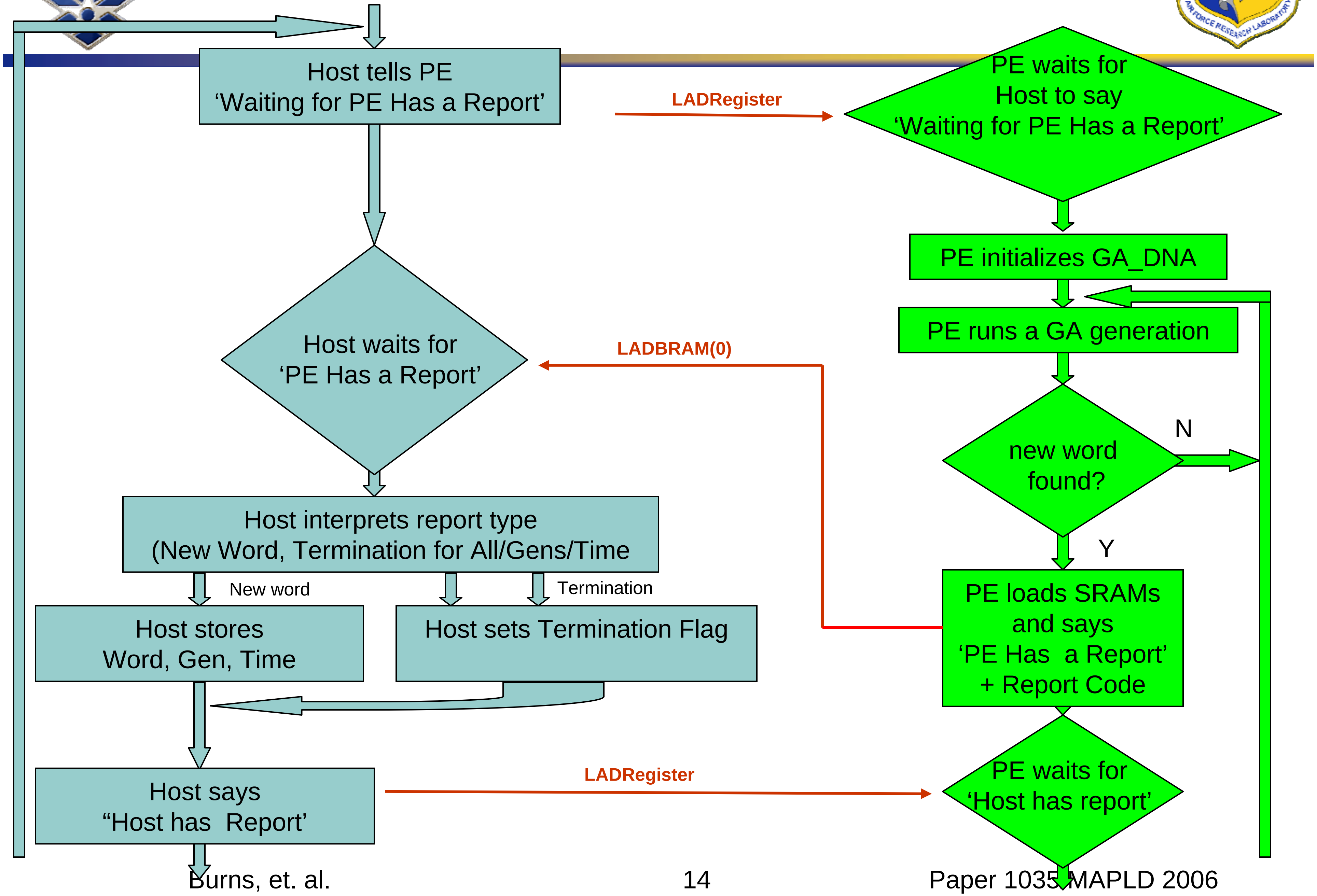
PE takes Pars from BRAM B side

Host waits for 'PE Has Pars'

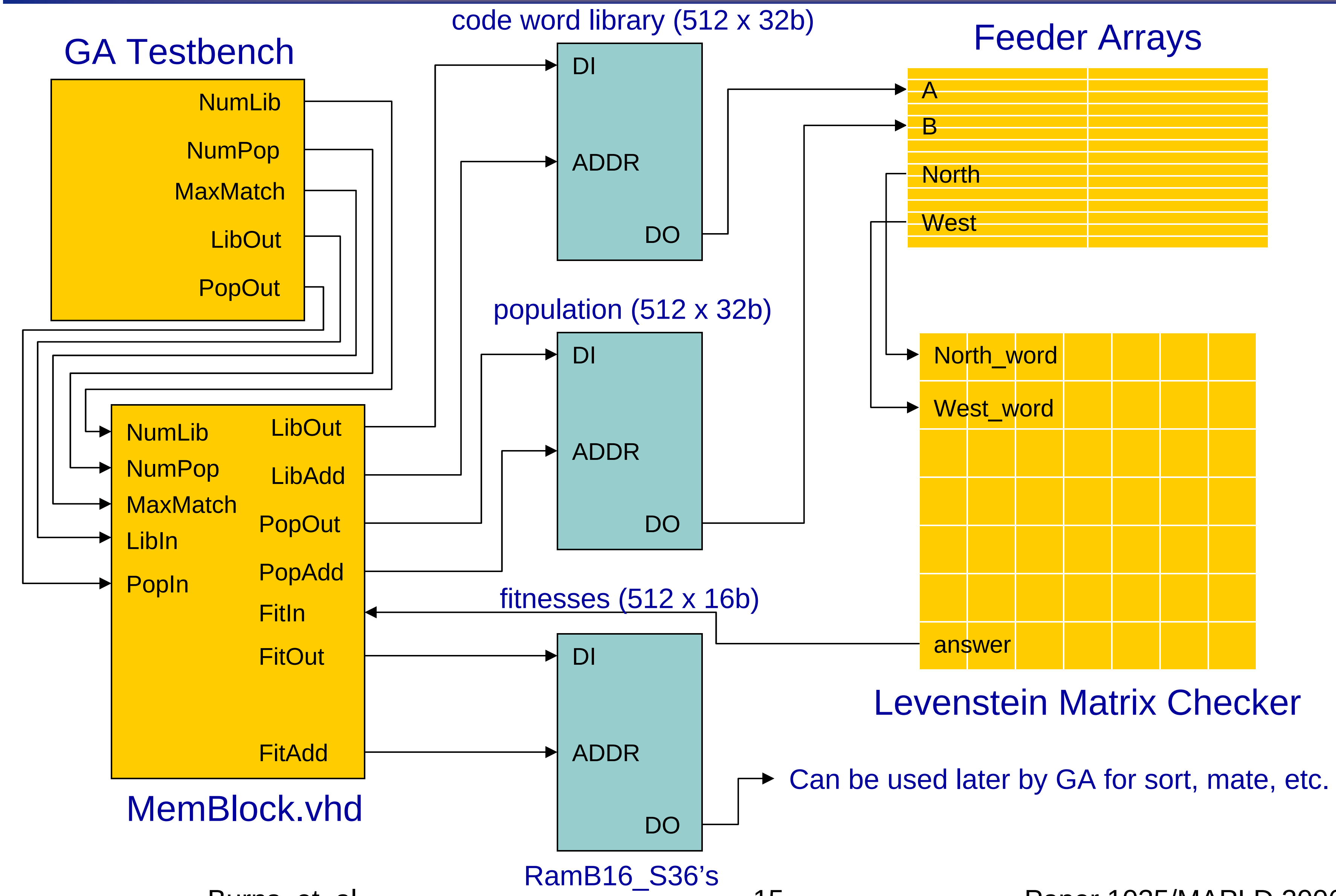
LAD_Bus_Out.Data_Out

PE says 'PE Has Pars'

Host and PE Process Interaction Time Line (2/4)



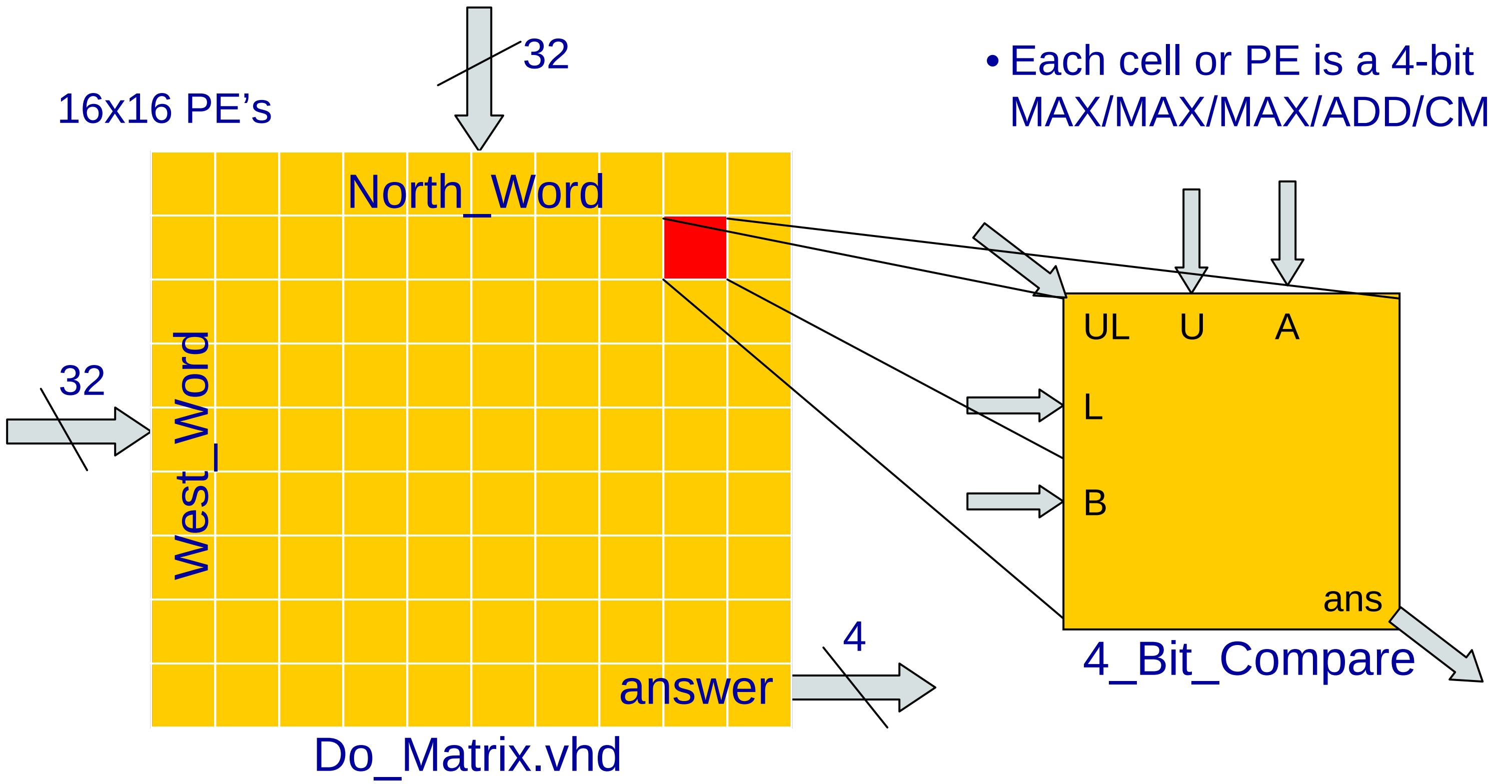
GA Core Data Path - Fitness Evaluator



Can be used later by GA for sort, mate, etc.

Levenstein Matrix Calculation Ripple Through 2D Pipeline Array

- Each cell or PE is a 4-bit MAX/MAX/MAX/ADD/CMP circuit



- **{U, L, UL} signals wired from adjacent cells' 'ans' output registers.**
- **{A, B} shift registers pass North and West Word 2bit base tokens down cols and across rows.**
- **{U, L, UL} connect to 0's depending on location on edges**

Resources and Performance:

Number of Slices:	**4273**	**out of**	**33792**	**12%**
Number of Slice Flip Flops:	**1**	**out of**	**67584**	**0%**
Number of 4 input LUTs:	**7593**	**out of**	**67584**	**11%**
Number of bonded IOBs:	**69**	**out of**	**1104**	**6%**

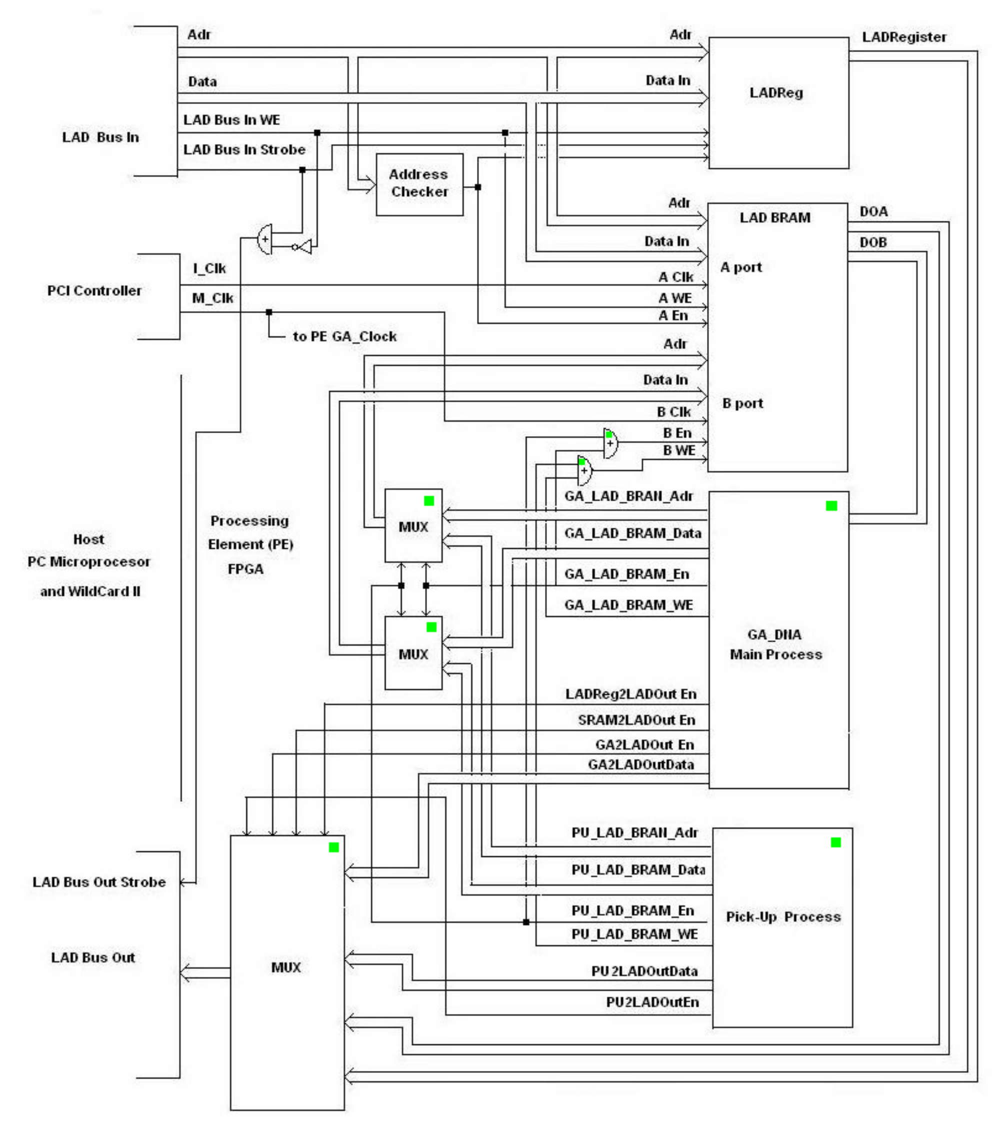
Estimated Delay: 185ns (5.4 MHz)

- **Much slower than 10ns target.**

GA/DNA Application Core Datapath

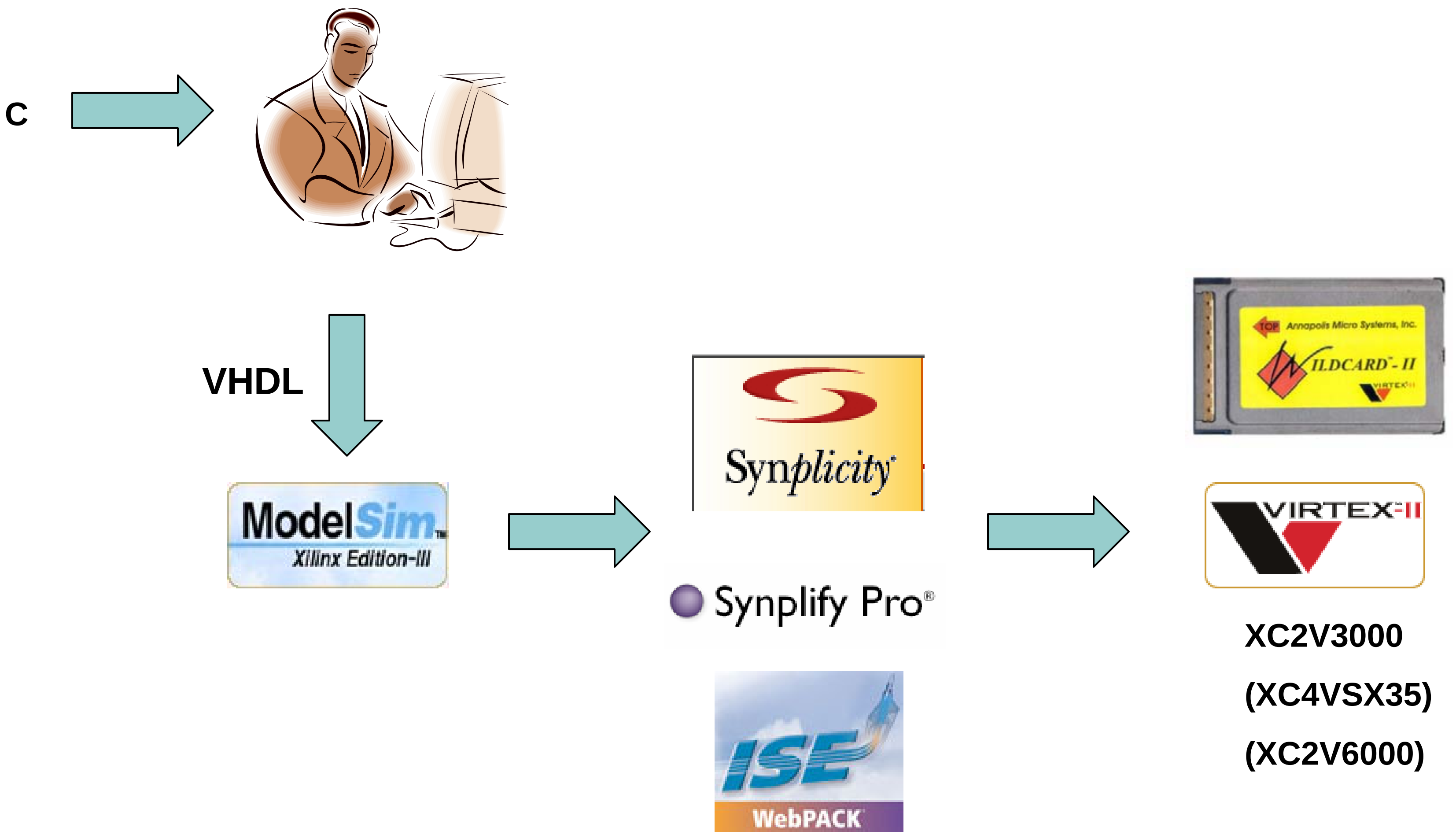
Platform: Notebook PC Host/WildCard II FPGA Processing Element (PE)

Software Template: LAD Interface Example



🟩 added or modified

Design Tool Paths



C

VHDL

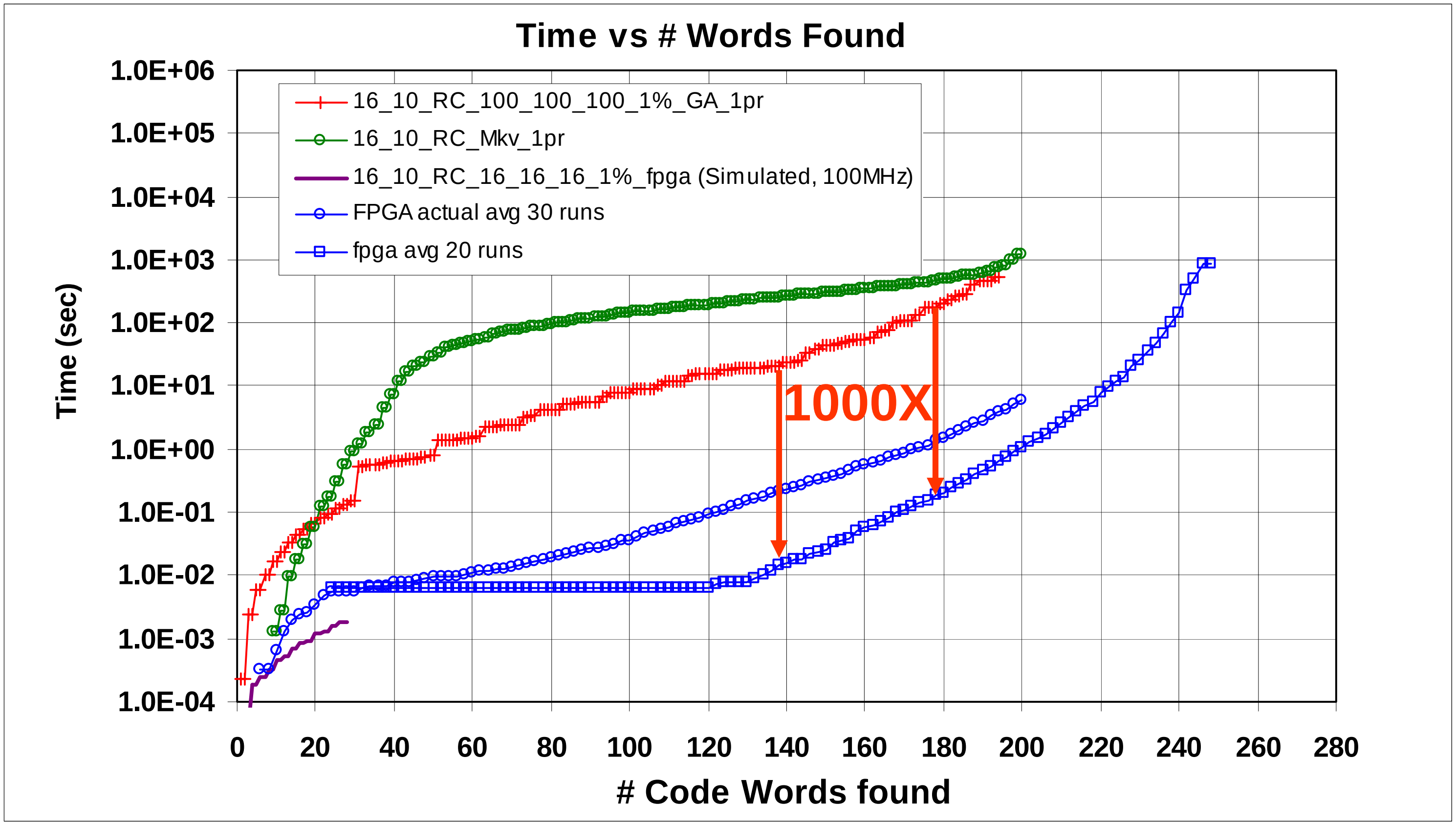
XC2V3000

(XC4VSX35)

(XC2V6000)

Average Performance Over Multiple Runs

Markov (16 runs), Software GA in C on PC (30 runs), Hardware GA (30 runs), Hardware GA + Exh. Search (20 runs)



- lower FPGA results are GA for 300sec or 120words, followed by Exhaustive Search to find all words remaining words, with 2 fitness evaluators, 100MHz on WildCard_II
- real speed-up is now ~ 1000X with 1 FPGA chip
- Simulated (ModelSim) hardware curve is for 100MHz, with one 1 clock systolic array
- upper actual FPGA v1 is 30MHz, one 2 clock SA, GA for 300sec followed by Exhaustive Search (ES) with 2 fitness evaluators to find all words remaining words
- lower actual FPGA v2 is 100MHz, GA for 300s or 240w one 2 clk SA + Exh Srch 100MHz, two 2 clk SA to optimal, with initial 22 word set passed in, checking whole pop each mutation

File I/O to enable extension of partial Code Word Libraries

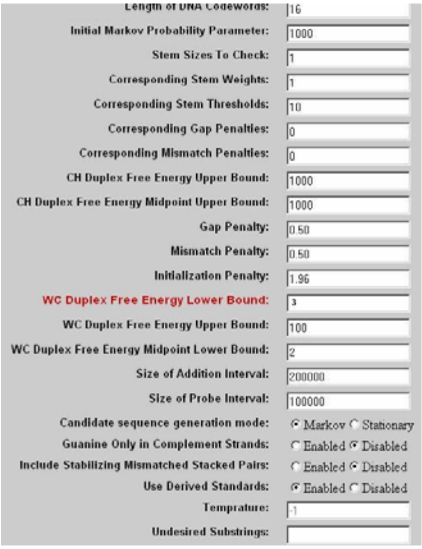
file_in.txt

GGTCTCATCTACATTC
CATGTATCACATGCCA

file_out.txt

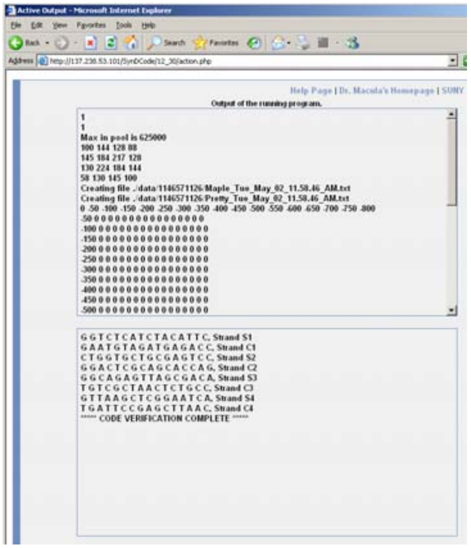
GGTCTCATCTACATTC
CATGTATCACATGCCA
CTGGTGCTGCGAGTCC
GGCAGAGTTAGCGACA
GTTAAGCTCGGAATCA
TATGCGTCACGTACGA

Codes are verified by SynDCode at
http://s53n101.academic.geneseo.edu/



Length of DNA Codewords:	16
Initial Markov Probability Parameter:	1000
Stem Sizes To Check:	1
Corresponding Stem Weights:	1
Corresponding Stem Thresholds:	10
Corresponding Gap Penalties:	0
Corresponding Mismatch Penalties:	0
CH Duplex Free Energy Upper Bound:	1000
CH Duplex Free Energy Midpoint Upper Bound:	1000
Gap Penalty:	0.50
Mismatch Penalty:	0.50
Initialization Penalty:	1.96
WC Duplex Free Energy Lower Bound:	3
WC Duplex Free Energy Upper Bound:	100
WC Duplex Free Energy Midpoint Lower Bound:	2
Size of Addition Interval:	200000
Size of Probe Interval:	100000
Candidate sequence generation mode:	● Markov ○ Stationary
Guanine Only in Complement Strands:	○ Enabled ● Disabled
Include Stabilizing Mismatched Stacked Pairs:	○ Enabled ● Disabled
Use Derived Standards:	● Enabled ○ Disabled
Temprature:	-1
Undesired Substrings:	

Output of FPGA version
verified by SynDCode



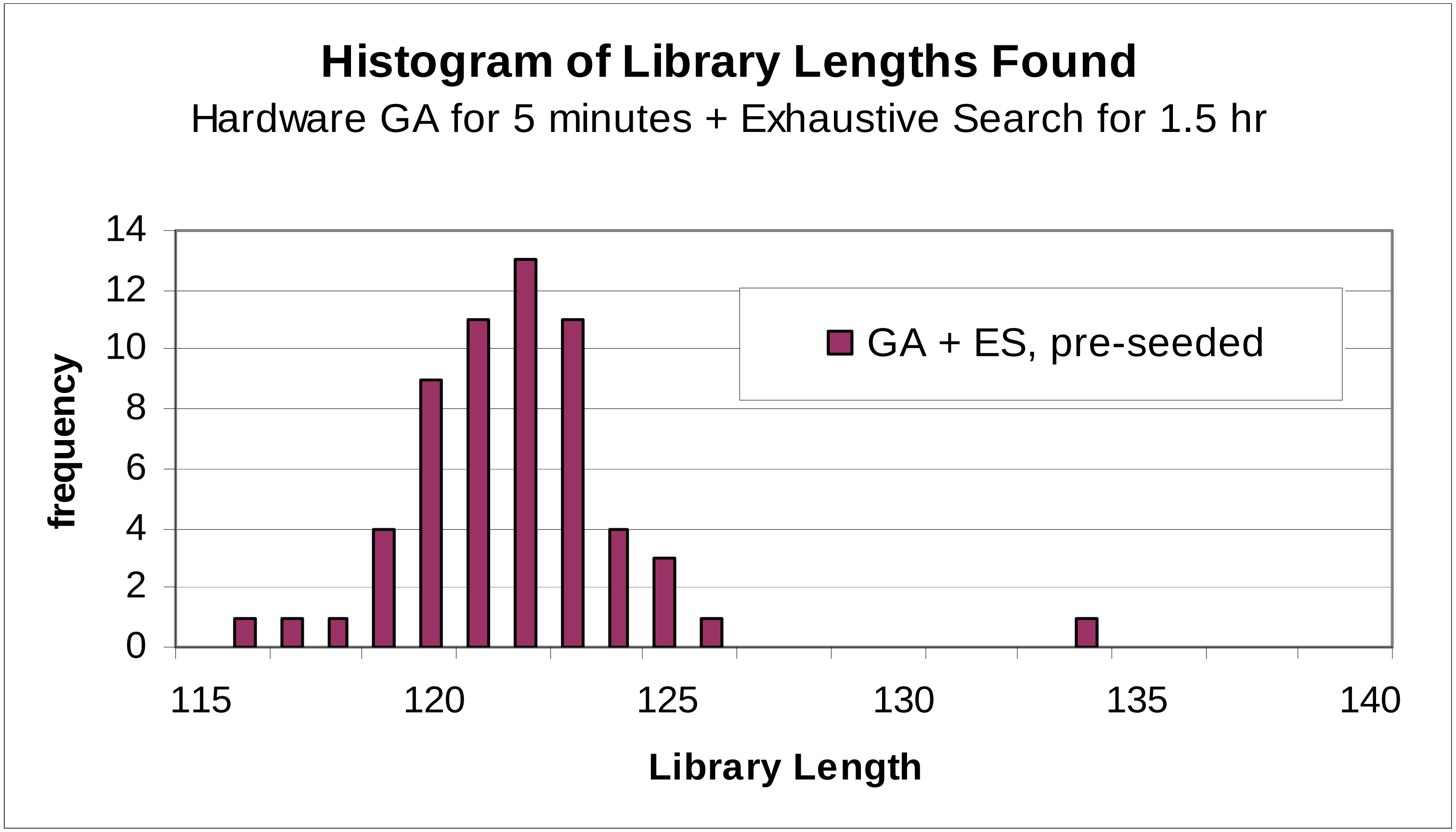
Active Output - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://137.238.53.101/SynDCode/12_30/action.php
Help Page | Dr. Macula's Homepage | SUNY

Output of the running program.

```
1
1
Max in pool is 625000
100 144 128 88
145 184 217 128
130 224 184 144
58 130 145 100
Creating file ./data/1146571126Maple_Tue_May_02_11.58.46_AM.txt
Creating file ./data/1146571126Pretty_Tue_May_02_11.58.46_AM.txt
0 -50 -100 -150 -200 -250 -300 -350 -400 -450 -500 -550 -600 -650 -700 -750 -800
-50 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-150 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-200 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-250 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-300 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-350 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-400 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-450 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-500 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
G G T C T C A T C T A C A T T C, Strand S1
G A A T G T A G A T G A G A C C, Strand C1
C T G G T G C T G C G A G T C C, Strand S2
G G A C T C G C A G C A C C A G, Strand C2
G G C A G A G T T A G C G A C A, Strand S3
T G T C G C T A A C T C T G C C, Strand C3
G T T A A G C T C G G A A T C A, Strand S4
T G A T T C C G A G C T T A A C, Strand C4
***** CODE VERIFICATION COMPLETE *****
```

Hardware GA + Exhaustive Search Average and Largest Library Results



- **Hardware GA: 60 tests = 90 hrs actual time**
- **Equivalent time on 30 node cluster would be 4 months**

Speed-up and Resources for Different Platforms

	Speed-up	Resources
GA and Fitness Function (FF) on PC 0.2us GA +9.8us FF (complete)	1 **(0.2us+9.8us)**	Low
Island Model GA and FF on Cluster (complete)	30X **(30 nodes)**	High
GA on PC FF on FPGA (VHDL, synthesis complete)	50x **10us/(0.2us+9.8ns)***	Low
GA and FF on one FPGA (current work)	1,000x **10us/10ns**	Low
GA and FF on HHPC (future work)	512,000x **(1,000x * 2 pops * 2xFPGA * 64 nodes * 2 chips/node)**	High

Collaborators

Professor Gary B. Lamont, Ph.D.
Department of Electrical & Computer Engineering
AFIT/ENG
WRIGHT-PATTERSON AFB
DAYTON, OH

Dr. Larry Merkle
Rose-Hullman Institute of Technology
Terre Haute, IN

Dr. Tony Macula
State University of New York at Geneseo
Geneseo, NY

Conclusions

- Genetic Algorithm performance on the DNA Code Word Library Generation problem is competitive with best known algorithms

- Hardware GA/DNA single FPGA version achieves extreme (1000x) speed-up with WildCard-II in a notebook PC

- While the distributed GA is generally applicable to many problem types, the hardware GA is especially suited to problems that have fitness functions involving a matrix of relatively simple integer-only or Boolean logic functions, especially if it can be efficiently implemented in a hardware systolic array.

Future work

- **Complete single chip FPGA multi-pop GA prototype**
- **New systolic array to incoporate improved GA fitness metric (weighted pair binding energy calculation)**
- **Do complete GA FPGA Core including selection, mating, decloning**
- **Transition new multi-pop GA core to cluster of FPGAs platform**
- **Identify additional optimization problem types that can use a hardware GA and Systolic Array fitness evaluator**

Genetic Algorithm Hardware References

S. Scott, A. Samal, and S. Seth, "HGA: A Hardware Based Genetic Algorithm", Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays, Monterey, CA, pp. 53-59, Feb. 1995.

- Problem: suite of 7 simple equation test case problems
- HW: Borg board using 2 XC4003's, 8K RAM, 3 XC4005's, 8MHz clock
- SW: Silicon Graphics 4D/440 with four MIPS R3000 CPUs at 33 MHz.
- Speed-up: 13-19x in terms of clock cycles, 100x thought to be possible.

P. Graham and B. Nelson, "Genetic algorithms in Software and in Hardware - a Performance Analysis of Workstation and Custom Computing Machine Implementations", 1996 Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, USA, April 1996, pp. 216 – 225.

- Problem: 25 city Traveling Salesman Problem
- SW: HP PA-RISC 125 MHz workstation
- HW: 4 SPLASH 2 FPGA system with 4 memories running at 11 MHz
- Speed-up: 4X in terms of execution time, 50X in terms of clock cycles
- ARPA contract to National Semiconductor in 1994

C. Aporntewan, and P. Chongstitvatana, "A hardware implementation of the Compact Genetic Algorithm", Proceedings of the 2001 Congress on Evolutionary Computation, 2001, Volume 1, May 2001, pp. 624 - 629 vol. 1.

- Problem: 32 bit one max problem
- SW: 200MHz Ultra Sparc II, SunOS.
- HW: Xilinx Virtex V1000FG680, 42 ns Tclk, (23.6Mhz)
- Speed-up: 1000X (0.15 sec vs 150 sec)

B.E. Wells, C. Patrick, L. Trevino, J. Weir, and J. Steincamp, " Applying a Genetic Algorithm to Reconfigurable Hardware – a Case Study", 2004 MAPLD, paper 169.

- Problem: 65 city Traveling Salesman Problem
- SW: 3.2 Ghz Intel Xeon processor with a large 3-level cache, Linux (Kernel 2.4.21 SMP), hosting a single user. GCC C compiler (version 3.2.2) with maximum supported level of optimization.
- HW: one Xilinx Virtex II 6000 in HC-36 Hypercomputer™ system from Star Bridge Systems, Inc. 66MHz clock.
- Speed-up: 11.4X in terms of execution time (using 8 function evaluators)

Genetic Algorithm Hardware References

M.K. Pakhira, R.K. De, “A Hardware Pipeline for Function Optimization Using Genetic Algorithms”, Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, Washington, DC, June 2005, pp. 949-956.

- Problem: suite of 7
- SW: classical (CGA), parallel GA (PGA)
- HW: pipelined GA (PLGA)
- Speed-up: 13-53x over CGA, 1.5-2.3x PLGA over PGA

G.M. Megson and I.M. Bland, “The Systolic Array Genetic Algorithm, An Example of a Systolic Arrays as a Reconfigurable Design Methodology”, Proceedings of the 1998 IEEE Symposium on FPGA’s for Custom Computing Machines, Apr. 1998, pp. 260-261.

- Problem: None (just doing the GA)
- SW: None
- HW: XC4036XL-09, 690 CLB’s
- Speedup: n/a 34.3ns per gene (28.3 MHz)

Code Word Library Design Problem References

From A. Brenneman and A E. Condon, “Strand Design for Bio-Molecular Computers”, (Survey Paper), Theoretical Computer Science, Vol. 287:1, 2002, pages 39-58:

[11] R. Deaton, R. C. Murphy, M. Garzon, D. R. Franceschetti, and S. E. Stevens, Jr., "Good encodings for DNA-based solutions to combinatorial problems," Proc. DNA Based Computers II, DIMACS Workshop June 10-12, 1996, L. F. Landweber and E. B. Baum, Editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 44, 1999, pages 247- 258.

[12] M. Garzon, R. Deaton, P. Neathery, D. R. Franceschetti, and S. E. Stevens, Jr., "On the encoding problem for DNA computing," Preliminary Proc. 3rd DIMACS Workshop on DNA Based Computers, June 23-25, 1997, pages 230- 237.

[13] R. Deaton, M. Garzon, R. C. Murphy, J. A. Rose, D. R. Franceschetti, and S. E. Stevens, Jr., "Genetic search of reliable encodings for DNA-based computation," Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors), Proceedings of the First Annual Conference on Genetic Programming 1996.

Proceedings of the 2007 IEEE Symposium on Computational
Intelligence in Bioinformatics and Computational Biology (CIBCB 2007)

Hybrid Architecture for Accelerating DNA Codeword Library Searching

Qinru Qiu Daniel Burns∗ Qing Wu Prakash Mukre
Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902
∗Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441
qqiu@binghamton.edu, Daniel.Burns@rl.af.mil, qwu@binghamton.edu, pmukre1@binghamton.edu

Abstract — **A large and reliable DNA codeword library is the key to the success of DNA based computing. Searching for the set of reliable DNA codewords is an NP-hard problem, which can take days on the state-of-art high performance cluster computers. This work presents a hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the discovery of DNA reverse complement, edit distance codes. Two applications of this architecture were implemented and evaluated, including a code generator that uses a genetic algorithm (GA) to produce nearly locally optimal codes in a few minutes, and a code extender that uses exhaustive search to produce locally optimum codes in about 1.5 hours for the case of length 16 codes. The experimental results demonstrate that the GA can find ~99% of the words in locally optimum libraries, and that the hybrid architecture provides more than 1000X speed-up compared to a software only implementation.**

I. INTRODUCTION

The DNA molecule is now used in many areas far beyond its traditional function. The first DNA-based computation was proposed by Adleman [1]. It demonstrates the effectiveness of using DNA to solve hard combinatorial problems. DNA molecules have also been used as information storage media and three dimensional structural materials for nanotechnology.

One of the major concerns of DNA computing is reliability. In DNA computing, the information is encoded as DNA strands. Each DNA strand is composed of short codewords. DNA computing is based on the *hybridization* process, which allows short single-stranded DNA sequences (i.e. *oligonucleotides*) to self-assemble to form long DNA molecules. The reliability of the computing is determined by whether the oligonuleotides can hybridize in a predetermined way. The key to success in DNA computing is the availability of a large collection of DNA codeword pairs that do not crosshybridize.

Various quality metrics have been proposed to guide the construction process [1]-[5]. The computation of these metrics dominates the run time of the code building process. While metrics based on the Gibbs energy and nearest neighbor thermodynamics and consideration of secondary structure formation give accurate measurement of hybridization, they are computationally costly, motivating the use of simplified metrics. One such metric is the *Levenshtein distance*, or the so-called *deletion-correcting* or *edit distance*, which has been used to construct DNA codes [6].

Regardless of the quality metric used, composing DNA codes is NP-hard because the number of potential codewords that must be searched increases exponentially with the length of the DNA codewords. Exhaustive checking is generally impractical for words of length greater than about 12 base pairs. Various algorithms have been proposed for building DNA codes, including the GA [7], Markov processes [8], and Stochastic methods [9]. Recent work [10] has shown that a hybrid GA blended with Conway's lexicode algorithm [11][12] achieves better performance than either alone in terms of generating useful codes quickly.

Search methods for DNA codes are extremely time-consuming, and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. Theory is lacking to provide tight upper bounds on the size of codeword sets, and the best known bounds are based on experiments. For example, the largest known reverse complement edit distance DNA codeword library (length 16, edit distance 10) consist of 132 pairs, composing such codes can take several days on a cluster of 10 G5 processors.

This paper focuses generally on speed-up techniques for the composition of reverse complement, edit distance, DNA codes of length 16, using a modified genetic algorithm that uses a locally exhaustive, mutation-only heuristic tuned for speed. Ongoing work to be reported elsewhere is addressing extensions to metrics involving nearest neighbor thermodynamics, a more general GA, codewords of length 32.

More specifically, we report a novel accelerator for DNA codeword composition that incorporates a hardware GA, hardware edit distance calculation, and hardware exhaustive search. Hardware exhaustive search extends an initial codeword library by doing a final scan across the entire universe of possible codewords, yielding a known locally optimum code. The proposed architecture consists of a host PC, a hardware accelerator implemented in reconfigurable logic on a *field programmable gate array* (FPGA) and a software program running in a host PC that controls and communicates with the hardware accelerator. The characteristics of the proposed architecture are as follows:

1. High performance. It utilizes programmable logic devices to enable pipelined and massively parallel processing of the data. Compared with software-only approaches, the new architecture can provide more than 1000X speed-up. For example, instead of 52 days, it only takes 1.5 hours to scan

the entire codeword space and to find all additional words that must be added to produce a locally optimum code.

2. High flexibility. The hardware accelerator can be configured by software program, and presently it can be run on a workstation PC equipped with an FPGA board, or on a notebook computer equipped with a PCMCIA FPGA card.
3. User friendly. The hardware accelerator is transparent to the user. Its control and access is accomplished by memory reads and writes based on a set of given protocols.

The remainder of this paper is organized as follows: Section II provides the necessary biological background and terminology. Section III introduces the problem definition and the genetic algorithm for DNA codeword search. Section IV gives the detailed information about how to accelerate the GA search fitness calculation. Sections V and VI provide details about the hybrid architecture and present a performance comparison between the software version of the GA and the best known (Markov) algorithm found in the literature, and early results on locally optimum codes. Finally, conclusions are given in Section VII.

II. BACKGROUND

The DNA molecule is a nucleic acid. It consists of two oligonucleotide sequences. Each sequence consists of a sugar-phosphate backbone and a set of *nucleotides* (also called *bases*) connecting with the backbone. The oligonucleotide sequence is oriented. One end of the sequence is denoted as 3' and the other as 5'. Only strands of opposite orientation can form stable duplex.

There are four types of bases: Adenine, Thymine, Cytosine, and Guanine. They are denoted briefly as A, T, C, and G respectively. Each base can pair up with only one particular base through hydrogen bonds: A+T, T+A, C+G and G+C. Sometimes we say that A and T are complementary to each other while C and G are complementary to each other. A Watson-Crick complement of a DNA sequence is another DNA sequence which replaces all the A with T or vise versa and replaces all the T with A or vise versa, and also switches the 5' and 3' ends. A DNA sequence binds most stably with its Watson-Crick complement. The stability of the binding is determined by the *free energy* of the hydrogen bonds.

The calculation of the free energy involves many considerations. In this paper, we only consider the first order effect, and use the number of Watson-Crick pairs between two DNA sequences to represent their bonding strength. Such approximation is widely adopted by the research works in DNA codeword design [6][12]. Furthermore, the DNA sequences of length 10 or greater are usually considered to be flexible [6]. Therefore, the binding strength of two DNA strands is measured by the length of the longest complementary subsequence (not necessarily contiguous) of one strand and the reverse of the other. For example, Figure 1 shows two DNA strands that bind with 5 Watson-Crick pairs. The longest complementary sequence between two flexible DNA strands, *A* and *B*, is the same as the *longest common sequence* (*LCS*) between *A* and $\overline{B}$ [6].

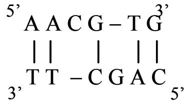


Figure 1 Binding between DNA strands.

III. PROBLEM FORMULATION AND OPTIMIZATION ALGORITHM

We consider each DNA codeword as a sequence of length *n* in which each symbol is an element of an alphabet of 4 elements. The longest common sequence between DNA strands *A* and *B* is denoted as *LCS*(*A*, *B*). In this work, we focus on searching for a set of DNA codeword pairs *S*, where S consists of a set of DNA strands of length *n* and their reverse complement strands e.g. $\{(s_1, \overline{s_1}), (s_2, \overline{s_2}), \ldots\}$, where $(s_1, \overline{s_1})$ denotes a strand and its Watson-Crick complement. The problem can be formulated as the following constrained optimization problem:

$$\max |S| \tag{1}$$

$$\text{s.t. } LCS(s_1, \overline{s_1}) \le \sigma, \quad \forall\, s_1 \in S\,, \tag{2}$$

$$LCS(s_1, s_2) \le \sigma, \forall s_1, s_2 \in S \tag{3}$$

$$LCS(s_1, \overline{s_2}) \le \sigma, \forall s_1, s_2 \in S\,, \tag{4}$$

where σ is a predefined threshold. Equation (1) indicates that our objective is to maximize the size of the DNA codeword library. The first constraint specifies that a DNA codeword in the library cannot bind with itself. The second and the third constraints specify that a DNA codeword in the library cannot bind with another library word or its Watson-Crick complement. Both of these two constraints must be satisfied because a DNA strand always occurs with its Watson-Crick complement.

A genetic algorithm (GA) is a stochastic search technique based on the mechanism of natural selection and recombination. Solutions, which are also called *individuals*, are evolved from generation to generation, with *selection*, *mating*, and *mutation* operators that provide an effective combination of exploration of the global search space. The *Island multi-deme* GA is a widely used parallel GA model in which the population is divided into several sub-populations and distributed on different processors. Each sub-population evolves independently for a few generations, before one or more of the best individuals of the sub-populations migrate across processors.

Although it is effective for many other optimization problems, we observed that selection and mating slowed the evolution of beneficial fitnesses in the population. Therefore, in this work, we propose a modified GA without mating. The approach is similar to Tulpan's [9], except that we start with an empty library, and a separate GA population of next word candidate individuals with random base content. Each individual in the population is a DNA codeword encoded as a binary string with length 2*n*, where *n* is the length of the codeword in bases. The four bases (A, T, C, G) are encoded as

(00, 01, 11, 10). Each DNA strand of length 16 can be represented as a 32 bit integer.

Given a codeword library *S*, the fitness of each individual *d* reflects how well the corresponding codeword fits into the current codeword library. Two values define fitness, *reject_num* and *max_match*. The *reject_num* is the number of codeword*s* in the library which satisfies the condition that $LCS(s,d) > \sigma$ or $LCS(\bar{s},d) > \sigma$. The *max_match* can be calculated as

$$\max(|LCS(d,\bar{d}) - \sigma|, |LCS(s,d) - \sigma|, |LCS(\bar{s},d) - \sigma|), \forall s \in S\ .$$

The codeword with lower fitness fits better in the library.

From equations (2)-(4) we know that a valid library word must have *reject_num* equal to 0. It is observed that adding a codeword with *reject_num* = 0 and *max_match* > 0 into the library will restrict the future growth of the library. Such codewords bind very weakly with other library words, but they are too far apart in the search space and interfere with closest packing. To maximize the library size, we want to select only those codewords that are "just good enough". To ensure this, we add another constraint to the optimization problem:

$$\max(LCS(s_1,s_2), LCS(s_1,\overline{s_2})) = \sigma, \forall s_1, s_2 \in S \qquad (5)$$

Therefore, only codewords with *reject_num* = 0 (which also implies *max_match* = 0) will be added into the library.

A traditional GA mutation function might randomly pick an individual in the population, randomly pick a pair of bits in the individual representing one of its 16 bases, and randomly change the base to one of the 3 other bases in the set of 4 possible bases. In the proposed algorithm, however, we randomly select an individual, but then to exhaustively check all of the 48 possible base changes. This is an attempt to speed beneficial evolution of the population by minimizing the overhead that would be associated with randomly picking this individual again and again in order to test those mutations. We also specify that if none of the 48 mutations were beneficial, one of them is selected at random. This enables the individual to remain in the population and possibly experience subsequent (multiple) mutations. Figure 2 gives the pseudo code for the modified mutation function.

When an individual in the population achieves a fitness of 0, it is added to the set of good codewords, and the selected individual in the population is replaced by a new random individual. The GA is allowed to run until one of three termination criteria is satisfied: the number of codewords in the set is as large as desired; the algorithm has run for a specified maximum number of generations; or the algorithm has run for a specified maximum amount of time. We store the codeword values and the elapsed time at which they are each found, in memory during a run, and we store that data to a disk file at the end of a run. We also calculate and store the average time at which the *i*th words are found across multiple runs to assess average performance.

```
Mutation( )
  //M is the set of mutated individuals;
  //L is the set of library codewords;
  Randomly select an individual s from initial population;
  $M = \Phi$;
  FOR i = 1 TO n
    $B = \{A, T, C, G\} - \{s[i]\}$; //B is the set of three nucleotides that is different from the ith nucleotide of s
    Generate three mutated individuals $\{s_1, s_2, s_3\}$ by replacing the ith nucleotide with one of the elements of B;
    $M = M \cup \{s_1, s_2, s_3\}$;
  END
  Evaluate the fitness for each $m \in M$;
  IF ($\exists m$, fitness(m) = 0) THEN $L = L \cup \{m\}$;
  ELSE //evolve the population by replacing the original individual with a new individual with better fitness
    Select the individual x which has the lowest (best) fitness and $x \in M$;
    IF fitness(x) < fitness(s) THEN replace s with x;
    ELSE replace s with a random individual from M;
  END
RETURN
```

Figure 2 Modified mutation algorithm.

IV. HARDWARE ACCELERATION OF LCS CALCULATION

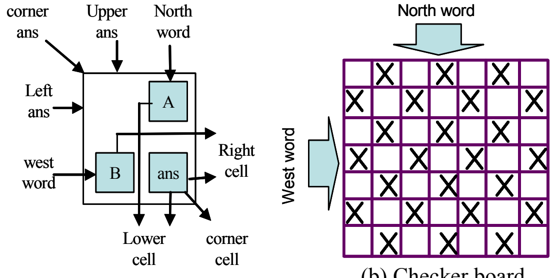
The most time consuming part of the proposed GA algorithm is to calculate the fitness value for each individual. Performance profiling of our software GA version showed that >98% of the computing time was spent calculating the *LCS* distance between DNA strands. The LCS distance is calculated using dynamic programming. Figure 3 gives the pseudo code of the algorithm. The intermediate results are stored in an *n*×*n* matrix, where *n* is the length of the DNA codeword in bases. The calculation starts at the top left corner of the matrix and the final result is the value calculated in the cell located at the bottom right corner. For DNA codewords with length 16, at least 256 operations are needed before we can obtain the final result. Therefore, the throughput of the software based LCS calculation is less than $1/n^2$.

```
LCS(a, b)
  Initialize lcs[0][i] and lcs[i][0], $0 \le i \le n-1$
  FOR i = 0 TO n-1 BEGIN
    FOR j = 0 TO n-1 BEGIN
      IF (a[i] = b[j]) THEN k = 1;
      ELSE k = 0;
      lcs[j][i] = max(lcs[j-1][i], lcs[j][i-1], lcs[j-1][i-1]+k);
    END
  END
```

Figure 3 LCS distance calculation.

The algorithm can be implemented using a 2D systolic array. The systolic array is an *n*×*n* matrix. Figure 4 (a) gives the structure of each cell in the matrix. Each cell consists of three registers: *A*, *B* and *ans*. For the cell at location (*i*, *j*), the registers *A* and *B* are used to store the *i*th nucleotide of one DNA codeword (north word) and the *j*th nucleotide of the other DNA codeword (west word) respectively. The register *ans* is used to store the intermediate result of the dynamic programming calculation. Each cell has five inputs. Two of the inputs connect to the register A and register B of the upper and left neighbor cells. The other three inputs connect to the *ans* registers of the upper, left and diagonal neighbor cells. In the present hardware version it takes two clock cycles for a cell to update its answer. In the first clock period, input registers *A* and *B* are updated, and in the second clock period, the cell output answer is calculated and the register *ans* is updated. In order to prevent ripple through operation, the cells in the even columns and even rows or odd columns and odd rows are synchronous to each other and operate as described above, but in the rest of the cells (which are also synchronous) the two operations are reversed, i.e. the *ans* output is calculated in the first clock period and the A and B inputs are updated in the second clock period.

The overall architecture of the 2D systolic array is shown in Figure 4 (b). The marked cells calculate their answers in the same clock cycle while the unmarked cells calculate their answers in the next clock cycle. In this way, the results propagate through the array diagonally. The final result is given by the *ans* register of the cell at the right bottom corner of the 2D array. It is easy to see that after a latency period that is required to fill the pipeline, the throughput of the systolic array is ½, i.e. 1 output result per 2 clock periods. When *n* increases, the throughput remains the same while the hardware cost increases, as long as the reconfigurable hardware chip has sufficient resources to implement a full *n*×*n* array of cells. Another detail is that the systolic array must be fed by an array of registers that delay the entry of the bases on the right of the North word and at the bottom of the West word. In effect, this synchronizes the presentation of those parts of the operand words with the diagonal waves of intermediate calculations in the cells that proceed from the upper left corner down and to the right through the array.



(a) Cell architecture

(b) Checker board architecture of 2D systolic

Figure 4 2D systolic array for LCS calculation.

We note that version of this array for words of length 32 vs. 16 would use 4X the resources, but clock at the same rate.

V. HYBRID ARECHITECTURE

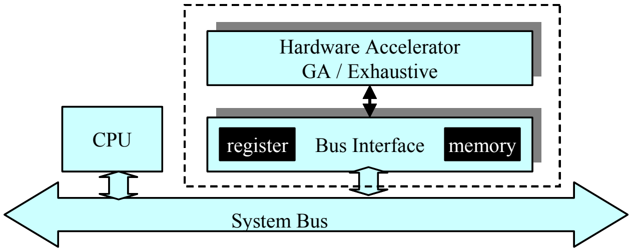


Figure 5 System architecture.

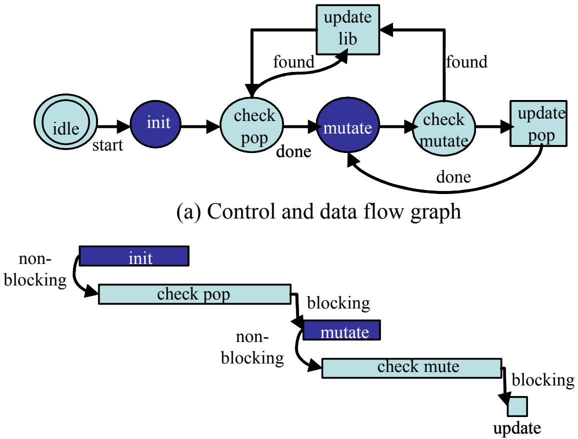
The proposed hybrid architecture consists of a host CPU, a hardware accelerator and a software program running on the host CPU. The host CPU and the hardware accelerator are connected via the system bus. Figure 5 shows the architecture of the system. In order to increase the portability of the design, we divide it into two modules: the bus interface and the hardware accelerator core. The bus interface module connects to the bus as a slave. It has a set of command registers and an information exchange memory, which can be accessed by both CPU and the hardware accelerator. For different bus architecture, a new bus interface must be developed.

A. Hardware acceleration for GA based codeword search

A two-level method is adopted to control the hardware accelerator. At the top level, the operations of the hardware accelerator are categorized into 7 states: {*idle*, *init*, *check_pop*, *mutation*, *check_mutate*, *update_pop*, *update_lib*}. In the *init* state, the hardware accelerator generates a random initial population, and sets up either an empty initial library, or reads an initial partial library from a disk file. In the *mutate* state, the hardware accelerator produces a population of 47 mutated individuals based on a chosen individual. The hardware accelerator calculates the fitness for all the individuals in the initial population, and in the mutated population, in the "*check_pop*" and "*check_mutate*" states, respectively. In the "*update_lib*" state, the hardware accelerator writes the newly discovered acceptable codewords into the library. In the "*update_pop*" state, the hardware accelerator writes the best (or a randomly chosen) mutated individual back to the working population.

Each state corresponds to an operation in the GA algorithm. Figure 6 (a) shows the control and data flow graph (CDFG) of the algorithm based on this state division. The "*update_lib*" and "*update_pop*" operations are one cycle operations because they only perform a memory write. All the other operations are multi-cycle operations, which again can be divided into several sub-states. When the top level state machine enters the corresponding state of a multi-cycle operation, the second level state machine is triggered.

We call an operation a *blocking operation* if its successors in the CDFG cannot start until this operation is done. Similarly, an operation is called *non-blocking operation* if its successors can start right after this operation started. The "*init*" and "*mutation*" operations are both non-blocking operations. While the hardware accelerator is generating the initial population and the mutated population, it is at the same time checking the fitness of the generated individual. The "*check_pop*" and "*check_mutate*" operations are blocking operations. Their successors, i.e. "*mutate*" and "*update_pop*", cannot start until they have been finished. Figure 6 (b) shows the scheduling of the operations.



(a) Control and data flow graph

(b) Scheduling of operations

Figure 6 Top level state machine controller.

A buffer is needed to pass the results of one operation to its successor. In particular, a first-in-first-out (FIFO) storage should be used as the output buffer of a non-blocking operation. However, the implementation of the FIFO is relatively easy in this design because the non-blocking operations are always faster than their successors. Therefore, it is not necessary to check the FIFO underflow condition. We use dual port memory as the output buffer for the design. Three memory blocks are used: Initial Population Memory (M_{pop}), Mutated Population Memory (M_{mutate}) and CodeWord Library Memory (M_{lib}). The input and output buffer of different operations are given in Table 1.

Table 1. The input/output buffer of operations.

operations	Input	Output
init	-	M_{pop}
check_pop	M_{pop}	M_{lib}
mutate	M_{pop}	M_{mute}
check_mutate	M_{mute}	M_{lib}
update_lib	M_{pop}	M_{lib}
update_pop	M_{mute}	M_{pop}

B. *Hardware software interface*

The hardware accelerator and the host CPU program run asynchronously. Four-way handshaking protocol is used to synchronize the communication between hardware and software, as shown in Figure 7. For example, when the hardware accelerator finds a new codeword, it raises the "PE_got_new_word" flag to the host program. After detecting this flag, the host program reads the new codeword then raises the "host_got_new_word" flag. After detecting this flag, the hardware accelerator then clears the "PE_got_new_word" flag and acknowledges the host program by raising the "PE_got_message" flag.

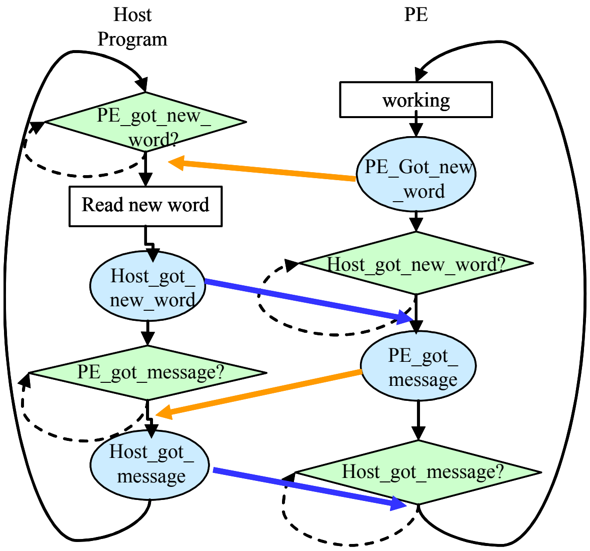


Figure 7 Hand-shaking between host and PE.

After detecting this flag, the host program then clears the "*host_got_new_word*" flag and acknowledges the hardware accelerator by raising the "*host_got_message*" flag, and continues. After detecting this flag, the hardware accelerator then clears the "*PE_got_message*" flag and continues. After the handshaking, the host program and the hardware accelerator work asynchronously until the host or hardware accelerator raises another message flag.

C. *Parallel GA*

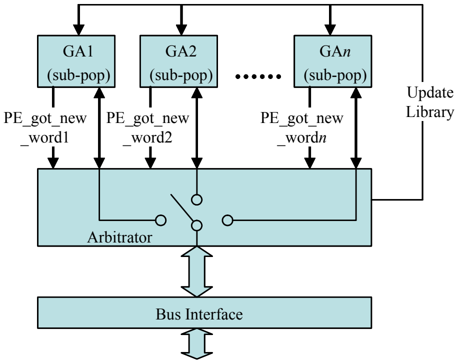


Figure 8 Hardware architecture for parallel GA.

The hardware accelerator discussed above uses about 12,263 LUTs (look-up-tables), which is only about 42% of the programmable resources in a Xilinx Virtex II 3000 FPGA and about 16% of the programmable resources in a Xilinx XC2VP70 FPGA. Therefore, we evaluated a further speed-up enhancement that involved implementing multiple parallel hardware accelerators on a single FPGA, as shown in Figure 8.

The system consists of *n* hardware accelerator modules, which are denoted as GA1~GA*n*, an arbitrator and a bus interface. The value of *n* is determined by the size of the FPGA. For example, *n* is 2 for the Virtex II 3000 FPGA and 5 for the XC2VP70. Each module implements the above mentioned genetic algorithm to search for the DNA codeword. They are independent to each other. The populations in different GA modules are initialized using different random seeds.

All the GA modules are connected to the bus interface through an arbiter. When a GA module finds a new codeword, it raises the "*PE_got_new_word*" flag and requests to be connected to the bus interface to communicate with the host. The arbiter broadcasts the new codeword to all other GA modules and raises the "*update_library*" flag. The GA module that receives the "*update_library*" request must terminate its current operation and go to "*update_lib*" state. If multiple GA modules raise the "*PE_got_new_word*" flag simultaneously, the arbiter must select one of them and invalidate the others. The decision is based on a fixed priority. The arbiter also connects the selected GA module that has found a new codeword with the bus interface to communicate with the host. If another GA module finds a new word, it must wait till the end of the current host-PE communication procedure to be connected to the bus interface. Figure 9 shows the state machine controller of the arbiter. The arbiter will be in the idle state after reset. When one of the GA modules raises the "*PE_got_new_word*" flag, the arbiter will go to the "*update_all_libraries*" state during which the arbiter raises the "*update_library*" flag. In the next clock period, it goes into the "*PE_communicating*" state during which the arbiter connects the GA module to the bus interface.

If the communication finishes before another GA module finds a new word, then the arbitrator goes back to the idle state. Otherwise, it first goes to the waiting state. After the communication is done, it goes to the "*update_all_libraries*" state and repeats the previous steps.

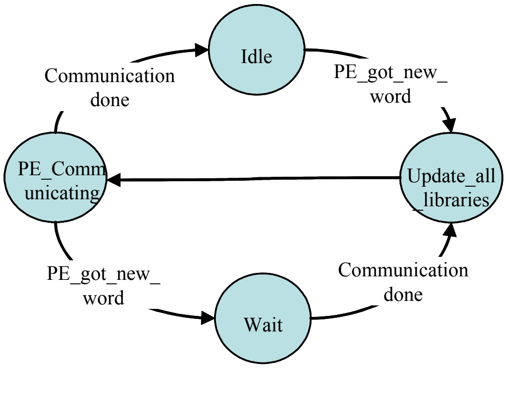


Figure 9 State machine controller of the arbitrator.

D. Hardware acceleration for exhaustive search

The effectiveness of the stochastic search starts decreasing when the search space increases and the solution space decreases. Therefore, as codewords are added to the library, the time required for the GA to find a new codeword increases exponentially. Furthermore, using stochastic search, we will never know whether still another new codeword can be added to the library. The only way to answer this question is by using exhaustive search, i.e. checking every possible codeword in the universe of all possible codewords. The complexity of exhaustive search increases linearly with the number of codewords already in the library. However, the complexity of exhaustive search also increases exponentially with the length of the codewords. As the name suggests, for a given initial library, the exhaustive search portion of the hybrid algorithm must scan the entire codeword space and find all remaining additional valid codewords that satisfy constraint equations (2)-(5). For DNA codewords of length 16, and for an initial library with 100 codewords, exhaustive search would take 52 days on a 2.0GHz Intel Xeon processor running a software fitness checker at 10 microseconds per check.

With small modification, we can implement the exhaustive DNA codeword search using hardware. The hardware accelerator for exhaustive codeword search consists of only one memory, which is used to store the codeword library, a 32 bit counter cycled from 0 to its maximum value to represent the potential new word, and two systolic array fitness checkers. For each codeword x, the calculation of $LCS(x,s)$ and $LCS(x,\bar{s})$, where $s \in S$, are performed simultaneously by the two fitness checkers. At 100Mhz clock frequency, the hardware accelerator takes about 1.5 hours to scan the entire ~4.3 billion codeword space for codewords of length 16, which is over 800 times faster than the workstation PC software only case. At the completion of exhaustive search we can say that a codeword set is *locally optimum*, in the sense that given the series of random numbers used to drive the stochastic GA in the early phase of building, no additional codewords can be added to increase the size of the library. To date, little data has been published in the literature on locally optimum edit distance codes of lengths greater than about 12 bases, and this hardware accelerator enables us to efficiently explore this aspect of the problem domain for the first time.

VI. EXPERIMENTAL RESULTS

A hardware accelerator that uses a stochastic GA to build DNA codeword libraries of codeword length 16 has been designed, implemented, and tested. The first version uses one fitness evaluator and is implemented on a single FPGA chip.

The design has actually been ported onto three different reconfigurable computing platforms, including a Xilinx XUP Virtex-II Pro evaluation board [13], a laptop computer with the Annapolis Wildcard FPGA board [14], and a desktop computer with the Annapolis Wildstar–II FPGA board. Different bus architectures are used to connect the hardware accelerator to the host CPU in each of the different platforms. The PLB bus is used in the Xilinx Virtex-II Pro evaluation board, while the PCMCIA card bus and PCI-X bus are used in the system with

WildStar and WildCard, respectively. The other difference among these platforms is the amount of resources available on the FPGA chips resident on the boards.

Table 2 shows the size of the reconfigurable logic and the on-chip memory for the three different computing platforms. The design is synthesized using Synplify from Synplicity. It uses 12,263 LUTs (look-up-tables), which is about 42% of the programmable resources in a Xilinx Virtex II 3000 FPGA. The hardware accelerator for exhaustive search of DNA codeword length 16 uses 21,733 LUTs, which is about 75% of Virtex II 3000 FPGA.

Table 2 Available reconfigurable logic and on-chip memory resources of different platforms.

Computing platform	FPGA	Logic Cells	BRAMs (kb)	PPCs
XUP eval. board	XC2VP30	30,816	2,448	2
WildCard-II	Xilinx Virtex II 3000	28,672	1,728	0
WildStar Pro	XC2VP70	74,448	5,904	2

Figure 10 shows a comparison of the average performance of the GA based codeword search algorithm running in software on a single workstation processor (upper curve) and the hardware accelerated hybrid architecture (lower line). The performance is measured in terms of the time it takes to build a large library. Less time is better, so the lower curve is better than the upper curve. In this plot the x axis is codewords found, where each codeword consists of a strand and its reverse complement. The GA is a stochastic algorithm, so each point in the curves is the average over multiple runs of the times taken to find the # of codewords on the x axis. For these experiments we set n and σ to be 16 and 10 respectively. The upper curve for the software version was run on one workstation with 1 P4 processor. The lower curve for the hardware GA was run with a 100MHz FPGA clock frequency.

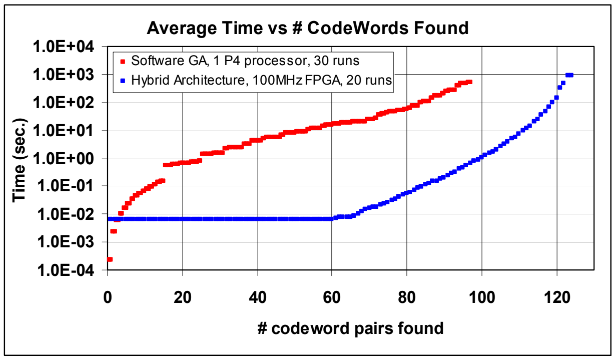


Figure 10 Comparison of average performance.

Compared to the software only implementation, the hardware accelerator running at 100MHz provides approximately a 1000X speed-up. The speed-up of the hardware versions is due to the parallel and pipelined architecture of the hardware. If we were able to increase either the number of fitness calculating arrays a we would expect almost linear speed-up ($a/0.98$). Also, based on previous work [15] that used a distributed Island Model GA run on a cluster of workstations, we would expect linear speed-up as the number of distributed GA populations p is increased.

Figure 11 shows a comparison of the best performance among software GA, and two versions of the hardware GA.

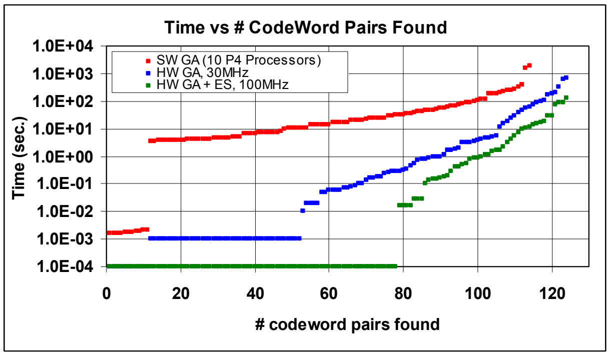


Figure 11 Comparison of best performance.

The top red curve for the distributed software multi-deme GA was run on a cluster using 10 P4 processors. The inter-processor communication is implemented using MPI (message passing interface). The middle blue curve for the hardware GA was run on the Annapolis Wildcard-II in a P3 notebook PC with a 30MHz FPGA clock frequency. The lower magenta curve for the hardware GA with exhaustive search was run on a Wildcard board in a P4 workstation with a 100MHz FPGA clock frequency. The later run was set up to run the GA until 240 words were found, and then switch to exhaustive search, after which 8 more words were found.

We also used the exhaustive search version of the hardware accelerator to investigate the average size of locally optimum codeword libraries that can be built, and the efficacy of the GA for building them. Figure 12 shows the distribution of the size of local optimal DNA codeword libraries that were generated by running hardware GA for 300 seconds followed by hardware exhaustive search. The results show that the size of the local optimal DNA codeword library follows a normal distribution with mean of about 122 codewords (word/word' pairs). The experiment consists of 60 tests, which took about 90 hours. The equivalent test on a 30 workstation cluster would have taken about 3000 hours (4 months).

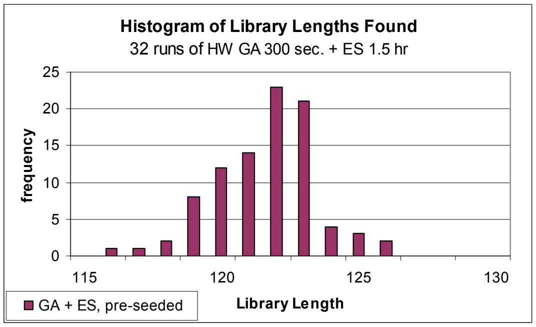


Figure 12 Size of local optimal DNA codeword libraries built with 300sec. GA plus exhaustive search.

Figure 13 shows data from a second experiment involving 32 runs of GA for 600 sec. followed by exhaustive search, in terms of the size of the library built during the GA phase (red) and the number of words added by exhaustive search (green).

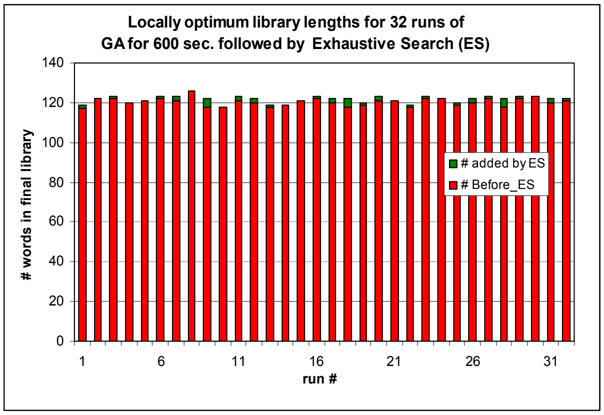


Figure 13 Sizes of Libraries built with 600 sec. GA followed by exhaustive search.

Figure 14 shows a histogram of the # of words added by exhaustive search for these runs. On average, the GA alone finds 120.4 words vs. 121.7 with GA + exhaustive search, or about 98.9% of the words that can be found.

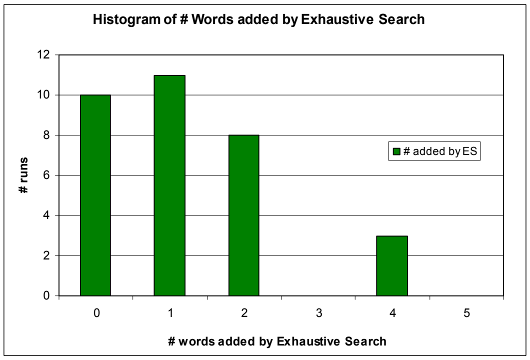


Figure 14 Histogram of # words added by Exhaustive Search for the runs of Figure 13.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we propose a novel architecture for accelerating a GA based DNA codeword searching algorithm. Our preliminary results show that, using a new hybrid hardware/software implementation, we can speedup the DNA codeword search procedure by more than 1000X. We have also described a hardware exhaustive search extension that can produce known locally optimum codes. In the future, we plan to extend the current architecture to implement a multi-deme GA on a single FPGA, a more general GA, more accurate techniques to measure the binding strength of DNA pairs, and a checker for codes word of at least length 32.

REFERENCES

[1] L. M. Adleman, “Molecular Computation of Solutions to Combinatorial Problems,” *Science*, vol. 266, pp. 1021-1024, November 1994.

[2] A. Brenneman and A. Condon, “Strand Design for Biomolecular Computation”, *Theoretical Computer Science*, vol. 287, pp.39-58, 2002.

[3] S.-Y. Shin, I.-H. Lee, D. Kim, and B.-T. Zhang, Multiobjective Evolutionary Optimization of DNA Sequences for Reliable DNA Computing”, *IEEE Transactions on Evolutionary Computation*, vol. 9(20), pp.143-158, 2005.

[4] F. Tanaka, A. Kameda, M. Yamamoto, and A. Ohuchi, Design of Nucleic Acid Sequences for DNA Computing based on a Thermodynamic Approach, Nucleic Acids Research, 33(3), pp.903-911, 2005.

[5] J. Santalucia, “ A Unified View of polymer, dumbbell, and oligonucleotide DNA nearest neighbor thermodynamics”, *Proc. Natl. Acad. Sci., Biochemistry*, pp. 1460-1465, February 1998.

[6] A. D’yachkov, P.L. Erdös, A. Macula, V. Rykov, D. Torney, C-S. Tung, P. Vilenkin and S. White, “Exordium for DNA Codes,” *Journal of Combinatorial Optimization*, vol. 7, no. 4, pp. 369-379, 2003.

[7] R. Deaton, M. Garzon, R.C. Murphy, J.A. Rose, D.R. Franceschetti, and S.E. Jr. Stevens, "Genetic search of reliable encodings for DNA-based computation," *Proceedings of the First Annual Conference on Genetic Programming*, pp. 9-15, July 1996.

[8] Bishop, M. , Macula, A. , Pogozelski, W. , and Rykov, V. , “DNA Codeword Library Design”, *Proc. Foundations of Nanoscience – Self Assembled Architectures and Devices, (FNANO)*, April 2005.

[9] Tulpan, D.C. , Hoos, H. , Condon, A. ,“Stochastic Local Search Algorithms for DNA Word Design”, *Eighth International Meeting on DNA Based Computers(DNA8)*, June 2002.

[10] S. Houghten, D. Ashlock and J. Lennarz, “Bounds on Optimal Edit Metric Codes”, *Brock University Technical Report # CS-05-07*, July 2005.

[11] O. Milenkovic and N. Kashyap, “On the Design of Codes for DNA Computing,” *Lecture Notes in Computer Science*, pp. 100-119, Springer Verlag, Berlin-Heidelberg, 2006.

[12] R. Brualdi, and V. Pless, “Greedy Codes,” *Journal of Combinatorial Theory Series A*, vol. 64, pp. 10-30, 1993.

[13] http://www.xilinx.com/

[14] http://www.annapmicro.com/

[15] D. Burns, K. May, T. Renz, and V. Ross, “Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis,” *MAPLD International Conference*, 2005.

Hardware Acceleration of Multi-deme Genetic Algorithm for the Application of DNA Codeword Searching

Qinru Qiu*, Daniel Burns**, Prakash Mukre*, Qing Wu*

*Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902

**Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441

qqiu@binghamton.edu, Daniel.Burns@rl.af.mil, pmukre1@binghamton.edu, qwu@binghamton.edu

ABSTRACT

A large and reliable DNA codeword library is key to the success of DNA based computing. Searching for sets of reliable DNA codewords is an NP-hard problem, which can take days on state-of-art high performance cluster computers. This work presents a hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the multi-deme genetic algorithm (GA) for the application of DNA codeword searching. The presented architecture provides more than 1000X speed-up compared to a software only implementation. A code extender that uses exhaustive search to produce locally optimum codes in about 1.5 hours for the case of length 16 codes is also described. The experimental results demonstrate that the GA can find ~99% of the words in locally optimum libraries. Finally, we investigate the performance impact of migration, mating and mutation functions in the hardware accelerator. The analysis shows that a modified GA without mating is the most effective for DNA codeword searching.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: *Performance attributes*

General Terms

Performance, Design

Keywords

DNA, Genetic Algorithm, Hardware Acceleration

1. INTRODUCTION

The DNA molecule is now used in many areas far beyond its traditional function. The first DNA-based computation was proposed and implemented by Adleman [1]. It demonstrates the effectiveness of using DNA to solve hard combinatorial problems. DNA molecules have also been used as information storage media and three dimensional structural materials for nanotechnology.

One of the major concerns of DNA computing is reliability. In DNA computing, the information is encoded as DNA strands. Each DNA strand is composed of short codewords. DNA computing is based on the hybridization process, which allows short single-stranded DNA sequences (i.e. oligonucleotides) to self-assemble to form stable double-stranded duplexes. The reliability of the computing is determined by whether the oligonuleotides can hybridize in a predetermined way. The key to success in DNA computing is the availability of a large collection of DNA codeword pairs that do not crosshybridize.

Various quality metrics have been proposed to guide the construction process [1]-[5]. The computation of these metrics dominates the run time of the code building process. While metrics based on the Gibbs energy and nearest neighbor thermodynamics and consideration of secondary structure formation give accurate measurement of hybridization, they are computationally costly, as a first step in this work we chose a simpler metric, the *Levenshtein distance*, or the so-called *deletion-correcting* or *edit distance*, which has also been used to construct DNA codes [6].

Regardless of the quality metric used, composing DNA codes is NP-hard because the number of potential codewords that must be searched increases exponentially with the length of the DNA codewords. Exhaustive checking is generally impractical for words of length greater than about 12 base pairs. Various algorithms have been proposed for building DNA codes, including the GA [7], Markov processes [8], and Stochastic methods [9]. Recent work [10] has shown that a hybrid GA blended with Conway's lexicode algorithm [11][12] achieves better performance than either alone in terms of generating useful codes quickly.

Search methods for DNA codes are extremely time-consuming, and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. Theory is lacking to provide tight upper bounds on the size of codeword sets, and the best known bounds are based on experiments. For example, the largest known reverse complement edit distance DNA codeword library (length 16, edit distance 10) consist of 132 pairs, composing such codes can take several days on a cluster of 10 G5 processors.

This paper focuses generally on speed-up techniques for the composition of reverse complement, edit distance, DNA codes of length 16, using a multi-deme genetic algorithm. We propose a FPGA (Field Programmable Gate Array) based hardware accelerator design which performs multi-deme parallel GA on a single chip. The hardware accelerator and the host PC communicate via the system bus, and an appropriate software interface controls communication between them. The proposed architecture provides more than 1000X speed-up compared to a software only implementation. A hardware based code extender that uses exhaustive search to produce locally optimum codes is also described. The code extender does a final scan across the entire universe of possible codewords and completes the

GECCO'07, July 7–11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-697-4/07/0007...$5.00.

codeword library generated from GA by adding any additional words that satisfy the specified constraints.

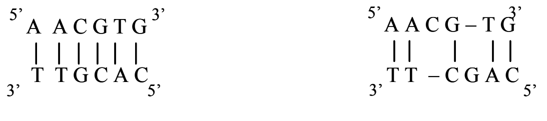
The remainder of this paper is organized as follows: Section 2 provides the necessary biological background and terminology. Section 3 introduces the problem definition and the genetic algorithm for DNA codeword search. Section 4 gives the detailed information about how to accelerate the GA fitness calculation. Sections 5 and 6 provide details about the hybrid architecture and some performance analysis of the design. Performance comparison between the hardware and the software version of the GA, and early results on locally optimum codes are also presented in Section 6. Final conclusions are given in Section 7.

2. BACKGROUND

The DNA molecule is a nucleic acid. It consists of two oligonucleotide sequences. Each sequence consists of a sugar-phosphate backbone and a set of *nucleotides* (also called *bases*) connecting with the backbone. The oligonucleotide sequence is oriented. One end of it is denoted as 3' and the other as 5'.

There are four types of bases: Adenine, Thymine, Cytosine, and Guanine. They are denoted briefly as A, T, C, and G respectively. Each base can pair up with only one particular base through hydrogen bonds: A+T, T+A, C+G and G+C. Sometimes we say that A and T (C and G) are complementary to each other. A Watson-Crick complement of a DNA sequence is another DNA sequence which replaces all the A with T or vise versa and replaces all the G with C or vise versa, and also switches the 5' and 3' ends. A DNA sequence binds most stably with its Watson-Crick complement and the structure they form is called *Watson-Crick* (*WC*) *duplex*. Figure 1 (a) shows an example of a WC duplex. We refer to the non-WC duplex as *crosshybridized* (*CH*) *duplex*. Figure 1 (b) shows an example of a CH duplex. Only WC duplexes are needed during DNA computing. Therefore, it is important to design the DNA codes such that a fixed temperature can be found that is well above the melting point of all CH duplexes and well below the melting point of all WC duplexes that can form from strands in the code.

Predicting crosshybridization involves many considerations. In this paper, we only consider the first order effect, and use the maximum number of possible Watson-Crick pairs between two sequences to represent their bonding strength. Such approximation is widely adopted by the research works in DNA codeword design [6][12]. The binding strength of two DNA strands is measured by the length of the longest complementary subsequence (not necessarily contiguous) of one strand and the reverse of the other. For example, Figure 1 (b) shows two DNA strands that bind with 5 Watson-Crick pairs. The length of the longest complementary sequence between two flexible DNA strands, A and B, is the same as the *length of the longest common sequence* (*LLCS*) between A and $\overline{B}$ [6], where $\overline{B}$ is the Watson-Crick complement of B.



(a) WC duplex (b) CH duplex

Figure 1 Binding between DNA strands.

3. PROBLEM FORMULATION AND OPTIMIZATION ALGORITHM

We consider each DNA codeword as a sequence of length n in which each symbol is an element of an alphabet of 4 elements. The length of the longest common sequence between DNA strands A and B is denoted as $LLCS(A, B)$. In this work, we focus on searching for a set of DNA codeword pairs S, where S consists of a set of DNA strands of length n and their reverse complement strands e.g. $\{(s_1, \overline{s_1}), (s_2, \overline{s_2}), \ldots\}$, where $(s_1, \overline{s_1})$ denotes a strand and its Watson-Crick complement. The problem can be formulated as the following constrained optimization problem:

$$\max |S| \quad \text{such that} \tag{1}$$

$$LLCS(s_1, \overline{s_1}) \le \sigma, \ \ \forall s_1 \in S, \tag{2}$$

$$LLCS(s_1, s_2) \le \sigma, \forall s_1, s_2 \in S \tag{3}$$

$$LLCS(s_1, \overline{s_2}) \le \sigma, \forall s_1, s_2 \in S, \tag{4}$$

where σ is a predefined threshold. Equation (1) indicates that our objective is to maximize the size of the DNA codeword library. The first constraint specifies that a DNA codeword in the library cannot bind with itself. The second and the third constraints specify that a DNA codeword in the library cannot bind with another library word or its Watson-Crick complement. Both of these two constraints must be satisfied because a DNA strand always occurs with its Watson-Crick complement.

A genetic algorithm (GA) is a stochastic search technique based on the mechanism of natural selection and recombination. Solutions, which are also called *individuals*, are evolved from generation to generation, with *selection*, *mating*, and *mutation* operators that provide an effective combination of exploration of the global search space. The *Island multi-deme* GA is a widely used parallel GA model in which the population is divided into several sub-populations and distributed on different processors. Each sub-population evolves independently for a few generations, before one or more of the best individuals of the sub-populations migrate across processors. In this work, the single point cross-over mating operator is used.

Each individual in the population is a DNA codeword encoded as a binary string with length $2n$, where n is the length of the codeword in bases. The four bases (A, C, G, T) are encoded as (00, 01, 10, 11). Each DNA strand of length 16 can be represented as a 32 bit integer. Given a codeword library S, the fitness of each individual d reflects how well the corresponding codeword fits into the current codeword library. Two values define the fitness, *reject_num* and *max_match*. The *reject_num* is the number of codeword*s* in the library which satisfy the condition that $LLCS(s,d) > \sigma$ or $LLCS(\overline{s},d) > \sigma$. The *max_match* can be calculated as

$$\max(\left|LLCS(d,\overline{d}) - \sigma\right|, \left|LLCS(s,d) - \sigma\right|, \left|LLCS(\overline{s},d) - \sigma\right|), \forall s \in S.$$

The codeword with lower fitness fits better in the library.

From equations (2)-(4) we know that a valid library word must have *reject_num* equal to 0. It is observed that adding a codeword with *reject_num* = 0 and *max_match* > 0 into the library will restrict the future growth of the library. Such codewords bind very weakly with other library words, but they are too far apart in the search space and interfere with closest packing. To maximize the library size, we want to select only those codewords that are "just good enough". To ensure this, we add another constraint to the optimization problem:

$$\max(LLCS(s_1,s_2),LLCS(s_1,\overline{s_2}))=\sigma, \forall s_1,s_2 \in S \qquad (5)$$

Therefore, only codewords with *reject_num* = 0 (which implies *max_match* = 0) will be added into the library.

A traditional GA mutation function might randomly pick an individual in the population, randomly pick a pair of bits in the individual representing one of its 16 bases, and randomly change the base to one of the 3 other bases in the set of 4 possible bases. In the proposed algorithm, however, we randomly select an individual, but then to exhaustively check all of the 48 possible base changes. This is an attempt to speed beneficial evolution of the population by minimizing the overhead that would be associated with randomly picking this individual again and again in order to test those mutations. We also specify that if none of the 48 mutations were beneficial, one of them is selected at random. This enables the individual to remain in the population and possibly experience subsequent (multiple) mutations. Figure 2 gives the pseudo code for the modified mutation function.

When an individual in the population achieves a fitness of 0, it is added to the set of good codewords, and the selected individual in the population is replaced by a new random individual. The GA is allowed to run until one of three termination criteria is satisfied: the number of codewords in the set is as large as desired; the algorithm has run for a specified maximum number of generations; or the algorithm has run for a specified maximum amount of time. We store the codeword values, the elapsed time at which they are each found in memory during a run, and store that data to a disk file at the end of a run. We also calculate and store the average time at which the *i*th words are found across multiple runs to assess average performance.

```
Mutation( )
    //$M$ is the set of mutated individuals;
    //$L$ is the set of library codewords;
    Randomly select an individual $s$ from initial population;
    $M = \Phi$;
    FOR $i$ = 1 TO $n$
        $B = \{A, T, C, G\} - \{s[i]\}$; //$B$ is the set of three nucleotides that is different from the $i$th nucleotide of $s$
        Generate three mutated individuals $\{s_1, s_2, s_3\}$ by replacing the $i$th nucleotide with one of the elements of $B$;
        $M = M \cup \{s_1, s_2, s_3\}$;
    END
    Evaluate the fitness for each $m \in M$;
    IF  ($\exists m$, fitness($m$) = 0)  THEN $L = L \cup \{m\}$;
    ELSE    //evolve the population by replacing the original individual with a new individual with better fitness
        Select the individual $x$ which has the lowest (best) fitness and $x \in M$;
        IF  fitness($x$) < fitness($s$)  THEN  replace $s$ with $x$;
        ELSE replace $s$ with a random individual from $M$;
    END
RETURN
```

Figure 2 Modified mutation algorithm.

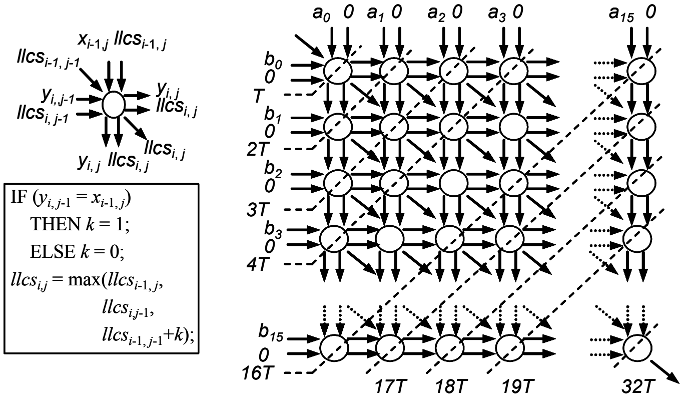
4. HARDWARE ACCELERATION OF LLCS CALCULATION

The most time consuming part of the proposed GA algorithm is to calculate the fitness value for each individual. Performance profiling of our software GA version showed that >98% of the computing time was spent calculating the *LLCS* between strands.

The *LLCS* is calculated using dynamic programming. Figure 3 gives the pseudo code of the algorithm. The intermediate results are stored in an $n \times n$ matrix, where n is the length of the DNA codeword in bases. The calculation starts at the top left corner of the matrix and the final result is the value calculated in the cell located at the bottom right corner. For DNA codewords with length 16, at least 256 operations are needed before we can obtain the final result. Therefore, the throughput of the software based *LLCS* calculation is less than $1/n^2$.

```
LLCS(a, b)
  Initialize llcs[0][i] and llcs[i][0], 0≤i≤n-1
  FOR $i$ = 0 TO $n$-1 BEGIN
    FOR $j$ = 0 TO $n$-1 BEGIN
      IF ($a[i] = b[j]$) THEN $k$ = 1  ELSE $k$ = 0;
      $llcs[j][i] = \max(llcs[j-1][i], llcs[j][i-1], llcs[j-1][i-1]+k)$;
    END
  END
END
```

Figure 3 LLCS distance calculation.



(a) Cell architecture (b) 2D systolic array

Figure 4 2D systolic array for LLCS calculation.

Many systolic algorithms has bee proposed to search for the longest common sequence (LCS) [16][17]. However, here we are only interested in finding out the length of the LCS, which is a much simpler problem. In this work, we implemented a 2D systolic array for the acceleration of LLCS calculation. The systolic array is an $n \times n$ array of identical cells. Figure 4 (a) gives the structure of each cell, including its input/output and the computation implemented. The computation is performed every other clock period. The overall architecture of the 2D systolic array as well as the data dependency and timing information are shown in Figure 4 (b). In order to prevent ripple through operation, the cells in the even columns and even rows or odd columns and odd rows are synchronous to each other and perform the computation in the same clock period. The rest of the cells are also synchronous to each other but perform the computation in the next clock period. In this way, the results propagate through the array diagonally. It is easy to see that after a latency period that is required to fill the pipeline, the throughput of the systolic array is ½, i.e. 1 output result per 2 clock periods.

It is interesting to note that as n increases, the hardware resource cost increases, but the throughput remains the same, as long as the reconfigurable hardware chip has sufficient resources to implement a full $n \times n$ array of cells. A version of this chip for words of length 32 is feasible. Another detail is that the systolic array must be fed by an array of registers that delay the entry of the bases on the right of word a and at the bottom of the word b. In effect, this synchronizes the presentation of those parts of the operand words with the diagonal waves of intermediate calculations in the cells that proceed from the upper left corner down and to the right through the array.

5. HYBRID ARECHITECTURE

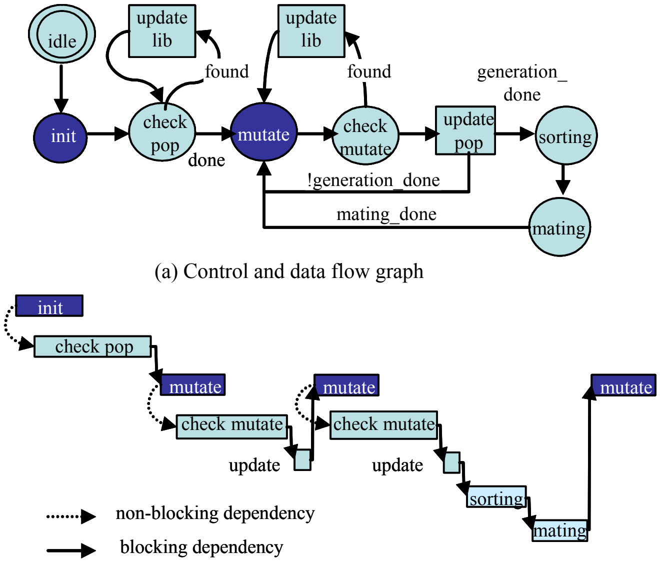
The proposed hybrid architecture consists of a host CPU, a hardware accelerator and a software program running on the host CPU. The host CPU and the hardware accelerator are connected via the system bus. In order to increase the portability of the design, we divide it into two modules: the bus interface and the hardware accelerator core. The hardware accelerator will also be called a processing element (PE) in the rest of the paper. The bus interface module connects to the bus as a slave. It has a set of command registers and an information exchange memory, which can be accessed by both CPU and the PE. For different bus architecture, a new bus interface must be developed.

5.1 Hardware acceleration for multi-deme parallel GA based codeword search

A two-level method is adopted to control the PE. At the top level, the operations of the PE are categorized into 9 states: {*idle*, *init*, *check_pop*, *mutation*, *check_mutate*, *update_pop*, *update_lib*, *sorting*, *mating*}. In the *init* state, the PE generates a random initial population, and sets up either an empty initial library, or reads an initial partial library from a disk file. In the *mutate* state, the PE produces a population of 47 mutated individuals based on a chosen individual. The PE calculates the fitness for all the individuals in the initial population, and in the mutated population, in the "*check_pop*" and "*check_mutate*" states, respectively. In the "*update_lib*" state, the PE writes the newly discovered acceptable codewords into the library. In the "*update_pop*" state, the PE writes the best (or a randomly chosen) mutated individual back to the working population. In the "*sorting*" state, the PE scans the entire population to pick the best k individuals. Two parents are randomly picked from these individuals when the PE is in the "*mating*" state and single-point cross-over is performed. A control flag is introduced which can be used to disable the sorting and mating functions in the PE.

Each state corresponds to an operation in the GA algorithm. Figure 5 (a) shows the control and data flow graph (CDFG) of the algorithm based on this state division. The "*update_lib*" and "*update_pop*" operations are one cycle operations because they only perform a memory write. All the other operations are multi-cycle operations, which again can be divided into sub-states. When the top level state machine enters the state of a multi-cycle operation, the second level state machine is triggered.

We call an operation a *blocking operation* if its successors in the CDFG cannot start until this operation is done. Similarly, an operation is called *non-blocking operation* if its successors can start right after this operation started. The "*init*" and "*mutation*" operations are both non-blocking operations. While the PE is generating the initial population and the mutated population, it is at the same time checking the fitness of the generated individual. The "*check_pop*" and "*check_mutate*", "*sorting*", and "*mating*" operations are blocking operations. Their following operations cannot start until they have been finished. Figure 5 (b) shows the scheduling of the operations.



(a) Control and data flow graph

(b) Scheduling of operations

Figure 5 Top level state machine controller.

A buffer is needed to pass the results of one operation to its successor. In particular, a first-in-first-out (FIFO) storage should be used as the output buffer of a non-blocking operation. However, the implementation of the FIFO is relatively easy in this design because the non-blocking operations are always faster than their successors. Therefore, it is not necessary to check the FIFO underflow condition. The output buffers are implemented using the FPGA built-in block memories. The block memories are dual port memories which can be read and written simultaneously. Three memory blocks are used: Initial Population Memory (M_{pop}), Mutated Population Memory (M_{mutate}) and CodeWord Library Memory (M_{lib}). The input and output buffers of different operations are given in Table 1.

Table 1. The input/output buffer of operations.

operations	Input	Output
init	-	M_{pop}
check_pop	M_{pop}	M_{lib}
mutate	M_{pop}	M_{mute}
check_mutate	M_{mute}	M_{lib}
update_lib	M_{pop}	M_{lib}
update_pop	M_{mute}	M_{pop}
sorting	M_{pop}	M_{pop}
mating	M_{pop}	M_{pop}

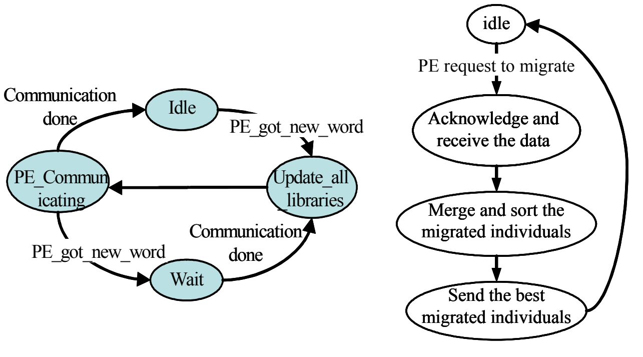
The PE and the host CPU program run asynchronously. Four-way handshaking protocol is used to synchronize the communication between hardware and software.

5.2 Parallel multi-deme GA

The PE discussed above uses about 12,263 LUTs (look-up-tables), which is only about 42% of the programmable resources in a Xilinx Virtex II 3000 FPGA and about 16% of the programmable resources in a Xilinx XC2VP70 FPGA. Therefore, we evaluated a further speed-up enhancement that involved implementing multiple parallel PEs on a single FPGA. The architecture supports the exchange of best individuals among PEs. Therefore, the overall system performs parallel multi-deme GA.

The system consists of *n* PE modules, which are denoted as GA1~GA*n*, an arbiter and a bus interface. The value of *n* is determined by the size of the FPGA. For example, *n* is 2 for the Virtex II 3000 FPGA and 5 for the XC2VP70. Each module implements the above mentioned genetic algorithm to search for the DNA codeword. They are independent of each other. The populations in different GA modules are initialized using different random seeds.

Communication and synchronization are two challenges that need to be addressed when designing a system that performs parallel GA. All the GA modules share the same bus interface. Codewords found by any one GA module must be harvested and passed to the other GA modules. In this design, all the GA modules are connected to an arbiter. When a GA module finds a new codeword, it raises the "*PE_got_new_word*" flag and requests to be connected to the bus interface to communicate with the host. The arbiter broadcasts the new codeword to all other GA modules and raises the "*update_library*" flag. The GA module that receives the "*update_library*" request must terminate its current operation and go to "*update_lib*" state. If multiple GA modules raise the "*PE_got_new_word*" flag simultaneously, the arbiter must select one of them and invalidate the others. The decision is based on a fixed priority. The arbiter also connects the selected GA module that has found a new codeword with the bus interface to communicate with the host. If another GA module finds a new word, it must wait till the end of the current host-PE communication procedure to be connected to the bus interface. Figure 6 (a) shows the state machine controller of the arbiter for library update.



(a) Controller for library update (b) Controller for individual Migration

Figure 6 State machine controller of the arbitrator.

In the multi-deme island GA, the best few individuals of each sub-population migrate periodically according to an interconnect configuration, e.g. around a ring in one direction. This procedure is also controlled by the arbiter. A separate state machine controller in the arbiter is developed for the migration procedure. Figure 6 (b) shows the state diagram of the migration controller. Periodically, the PE sends a migration request to the arbiter. The arbiter will acknowledge this request if its migration controller is in the idle state. After receiving the acknowledgement from the arbiter, the PE sends its best few individuals and their fitness values to the arbiter. These data are placed in a memory together with similar data received from other PEs. The arbiter sorts and picks the best *m* individuals, where *m* is the number of individuals to be migrated, and sends them back to the PE which started the request for migration. For the case of 2 PEs on a chip served by one arbiter, this is equivalent to a directed ring configuration. However, for the case of more than 2 PEs on a chip, this approach implements a local pooling, or all-to-all configuration. Above the chip level, the host is still free to implement any communication configuration among host nodes in a cluster with standard MPI.

5.3 Hardware acceleration for exhaustive search

The effectiveness of the stochastic search decreases when the size of the search space increases, or when the solution space decreases due to additional constraints. As codewords are added to the library, more library words must be checked against candidates, and the new words act as new constraints. As a result, the time required for the GA to find a new codeword increases exponentially. Furthermore, using stochastic search, we will never know whether still another new codeword can be added to the library. The only way to answer this question is by using exhaustive search, i.e. checking every possible codeword in the universe of all possible codewords. The complexity of exhaustive search increases linearly with the number of codewords already in the library. However, the complexity of exhaustive search also increases exponentially with the length of the codewords. As the name suggests, for a given initial library, the exhaustive search portion of the hybrid algorithm must scan the entire codeword space and find all remaining additional valid codewords that satisfy constraint equations (2)-(4). For DNA codewords of length 16, and for an initial library with 100 codewords, exhaustive search in software would take 52 days on a 2.0GHz Intel Xeon processor if checking a pair takes 10 microseconds.

With small modification, we can implement the exhaustive DNA codeword search using hardware. The hardware accelerator for exhaustive codeword search consists of only one memory, which is used to store the codeword library, a 32 bit counter cycled from 0 to its maximum value to represent the potential new word, and two systolic array fitness checkers. For each codeword *x*, the calculation of $LLCS(x,s)$ and $LLCS(x,\bar{s})$, where $s \in S$, are performed simultaneously by the two fitness checkers.

The hardware accelerator for exhaustive search of DNA codewords of length 16 uses 21,733 LUTs, which is about 75% of Virtex II 3000 FPGA. At 100Mhz clock frequency, the hardware accelerator takes about 1.5 hours to scan the entire ~4.3 billion codeword space for codewords of length 16, which is over 800 times faster than the workstation PC software only case. At the completion of exhaustive search we can say that a codeword set is *locally optimum*, in the sense that given the series of random numbers used to drive the stochastic GA in the early phase of building, no additional codewords can be added to increase the size of the library. To date, little data has been published in the literature on locally optimum edit distance codes of lengths greater than about 12 bases, and this hardware accelerator enables us to efficiently explore this aspect of the problem domain for the first time.

6. EXPERIMENTAL RESULTS

A hardware accelerator that uses a stochastic GA to build DNA codeword libraries of codeword length 16 has been designed, implemented, and tested. The first version uses one fitness evaluator and is implemented on a single FPGA chip.

The design has actually been ported onto three different reconfigurable computing platforms, including a Xilinx XUP Virtex-II Pro evaluation board [13], a laptop computer with the Annapolis Wildcard FPGA board [14], and a desktop computer with the Annapolis Wildstar–II FPGA board. Different bus architectures are used to connect the hardware accelerator to the host CPU in each of the different platformsThe other difference among these platforms is the amount of resources available on the FPGA chips resident on the boards.

The first set of experiments evaluates the performance impact of various parameters of the hardware multi-deme GA, including the size of the sub-population (*pop*), the percentage of mutation during each generation ($m\%$), the length of epoch (E) between migrations and the number of best individuals that migrate (n). The total number of mutations that are performed in each generation is calculated as $(m\% \times pop \times L)$, where L is the length of the codeword. For each mutation, the routine that is given in Figure 2 is executed. For this first set of experiments, the hardware implementations consisted of 2 parallel PEs that perform GA based codeword searching, each with one LLCS checker, and without exhaustive search.

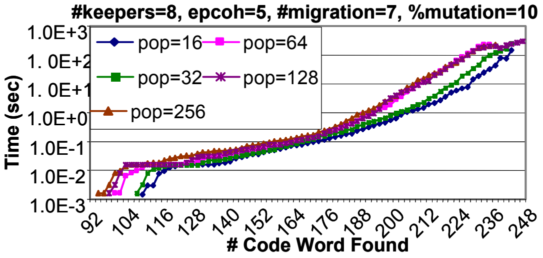


Figure 7 Effect of size of sub-population

We first ran the DNA codeword searching varying sub-population from 16 to 256. The number of keepers, the length of the epoch, number of migrated individuals, and the percentage mutation were fixed to be 8, 5, 7 and 10. Figure 7 shows a comparison of the average performance of those runs, in terms of the time it takes to build a large library. Less time is better, so the lower curve is better than the upper curve. In all the plots given by Figure 7-13, the x axis is the number of codewords found, where each codeword is either a strand or its reverse complement (a pair counts for 2). The GA is a stochastic algorithm, so each point in the curves is the average over 10 runs of the times taken to find the # of codewords on the x axis. For these experiments we set the length of the codewords n to be 16, and the permissible match (n-edit distance) σ to be 10. The experimental results show that with mating and migration enabled, a small population is superior to a large population in terms of search speed. This is because the most time consuming operation in mating and migration is pick up the best k individuals, which we call the number keepers. This does not require a full sort of the population, but even so, it is a sequential procedure that cannot be accelerated by a parallel architecture for typical population sizes. It takes more time to index through a larger population multiple times to find its best k individuals.

In the second experiment, we vary the percentage mutation from 1 to 25 to evaluate its impact on performance. The size of sub-population, the number of migrated individuals, the length of epoch and the number of keepers were fixed to be 64, 7, 5 and 8. Figure 8 shows a comparison of the average performance of different configurations. As we can see, the percentage mutation has a significant impact on the system performance.

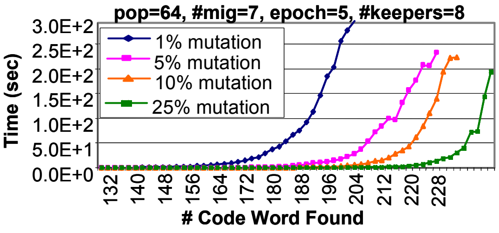


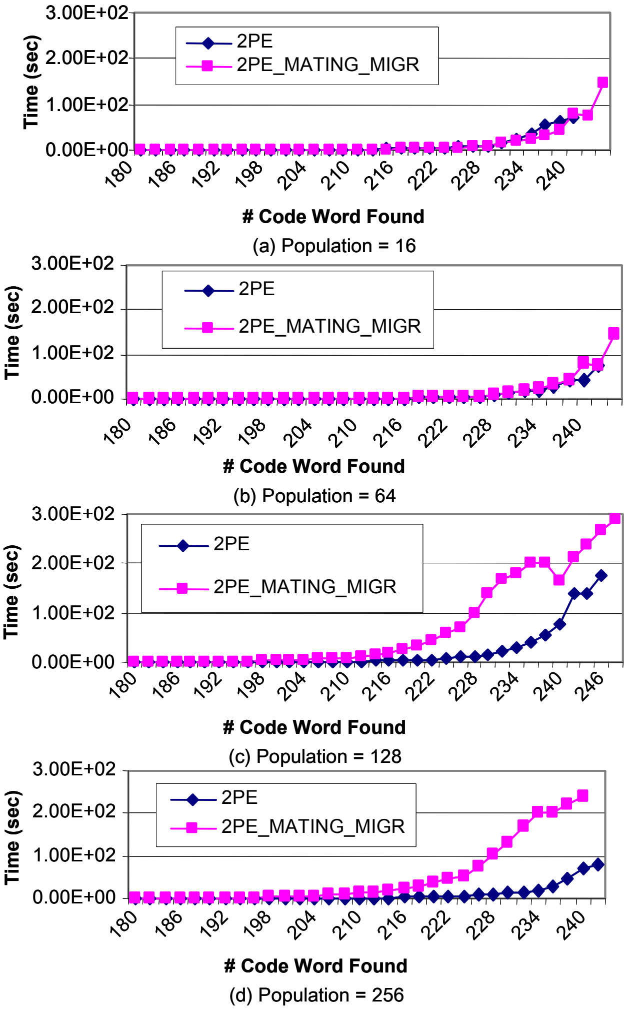
Figure 8 Effect of mutation

Higher percentage mutation leads to better performance. For example, to find 206 codewords, the hardware GA configured with 25% mutation is about 400X faster than the hardware GA configured with 1% mutation. This can partly be explained by the overhead of mating. When the size of population is fixed, the value of percentage mutation determines how many mutation operations will be performed between two mating operations. Because each mutation operation takes fixed amount of time, it also determines the frequency of mating operations. A higher percentage mutation implies less frequent mating, and thus, lower overhead from the sorting operation.

In the third and the fourth experiments, we vary the number of migrated individuals and the number of generations in the epoch between migrations, respectively, to evaluate the performance impact of these two parameters. However, the results show that there is little performance impact from the number of migrated individuals and the epoch length. Due to the space limit, we do not report this data in the paper.

The second set of experiments compares the performance of multi-deme hardware GA with and without mating and migration. Figure 9 shows a comparison of the average performance of GA with mating and migration versus GA without mating and migration when the size of sub-population varies from 16 to 256. The number of keepers, the length of the epoch, the size of migrated individuals and the percentage mutation are fixed to be 8, 5, 7 and 10. As we can see, overall, the parallel GA without mating and migration is more efficient than the parallel GA with mating and migration. The difference becomes more significant as the size of population increases. Again, this shows that the overhead of mating increases as the population size increases.

Figure 10 analyzes the data from this experiment in terms of the speed improvement of parallel GA without mating and migration, for different population sizes, normalized to the performance with population size 16. As we can see, at the beginning of the search, smaller populations find words faster, but as the number of codewords increases, the larger populations find word slightly faster. This effect may be due to the beneficial effect of processing more mutations in between pick-up operations at the end of generations (doing wider search) outweighs the negative effect of the overhead of the pickup operation that also increases with population size.



(a) Population = 16

(b) Population = 64

(c) Population = 128

(d) Population = 256

Figure 9 Effect of mating and migration.

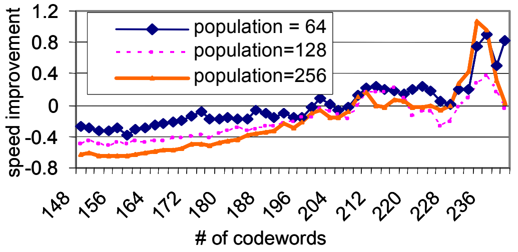


Figure 10 Effect of size of population in GA w.o. mating and migration.

Figure 11 shows the performance comparison between a single PE system and a 2 PE system. Both systems are configured with population size equal to 16 and both are running without mating and migration. As expected, the 2-PE system is about twice as fast as the one PE system

The next set of experiments compares the hardware GA with a software version of the GA, again without mating and migration, and with one PE is instantiated in the hardware.

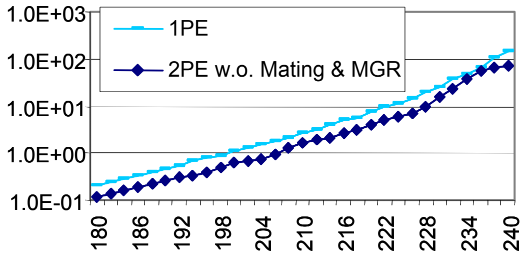


Figure 11 Performance comparison of single PE vs. 2-PE

Figure 12 shows a comparison of the average performance of the GA based codeword search algorithm running in software on a single workstation processor (upper curve) and the hardware accelerated hybrid architecture (lower line). The upper curve for the software version was run on one workstation with 1 P4 processor. The lower curve for the hardware GA was run with a 100MHz FPGA clock frequency.

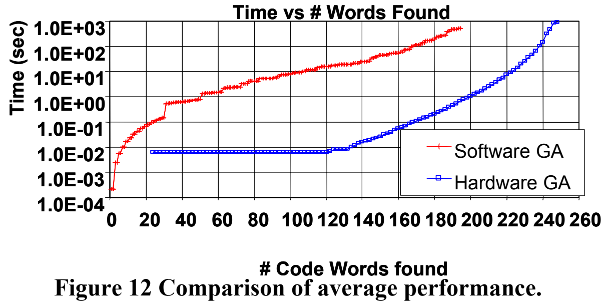


Figure 12 Comparison of average performance.

Compared to the software only implementation, the hardware accelerator running at 100MHz provides approximately a 1000X speed-up. The speed-up of the hardware versions is due to the parallel and pipelined architecture of the hardware. Based on previous work [15] we would expect almost linear speed-up (*a*/0.98) vs. the number of fitness calculators, and linear speed-up as the number of distributed GA populations *p* is increased.

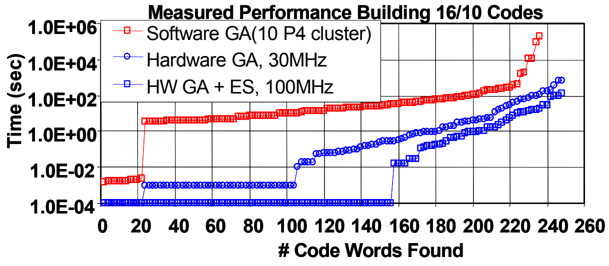


Figure 13 Comparison of best performance.

Figure 13 shows a comparison of the best performance to date of the software GA and the hardware GA. In this case, the top red curve for the distributed software multi-deme GA was run on a cluster using 10 P4 processors without mating, but with migration. The inter-processor communication is implemented using MPI (message passing interface). The middle blue curve for the hardware GA was run on the Annapolis Wildcard-II in a notebook PC with a 30MHz FPGA clock frequency, without mating and migration. The lower magenta curve for the hardware GA with exhaustive search was run on a Wildcard board in a P4 workstation with a 100MHz FPGA clock frequency, also without mating and migration (exhaustive search found 8 more words).

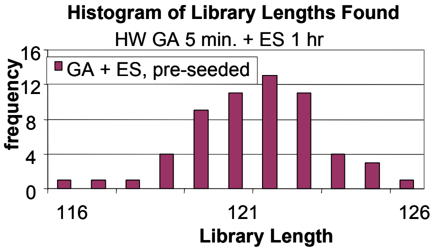


Figure 14 Size of local optimal DNA codeword libraries built with 300sec. GA plus exhaustive search.

In a final set of experiments, we used the exhaustive search version of the hardware accelerator to determine the average size of locally optimum codeword libraries that can be built, and the efficacy of the GA for building them. Figure 14 shows a histogram of the sizes of libraries generated by running hardware GA (without mating and migration) for 300 seconds followed by hardware exhaustive search. The results show that the size of the local optimal DNA codeword library follows approximately a normal distribution with mean of about 122 codewords (word/word' pairs). The experiment consists of 60 tests, which took about 90 hours. The equivalent test on a 30 workstation cluster would have taken about 3000 hours (4 months).

Figure 15 shows data from a second experiment involving 32 runs of the same hardware GA for 600 sec. followed by exhaustive search. The number of words found during the GA phase (red) and the exhaustive search phase (green) is highlighted.

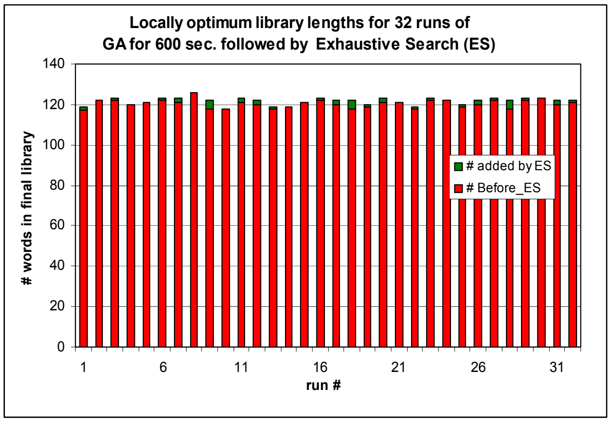


Figure 15 Sizes of Libraries built with 600 sec. GA followed by exhaustive search.

The GA phase alone finds an average of 120.4 words, and exhaustive search raises the number found to 121.7. So, GA alone found about 98.9% of the words that can be found.

7. CONCLUSIONS AND FUTURE WORK

In this work, we propose a novel architecture for accelerating a multi-deme parallel GA based DNA codeword searching algorithm. Our preliminary research results show that, using a new hardware and software hybrid implementation, we can speedup the DNA codeword search procedure by more than 1000X. We have also described a hardware exhaustive search extension that can produce known locally optimum codes. In the future, we plan to extend the current architecture to incorporate thermodynamics based metrics for estimating the binding strength of DNA pairs, and a checker for codes word of at least length 32.

8. References

[1] L. M. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, vol. 266, pp. 1021-1024, November 1994.

[2] A. Brenneman and A. Condon, "Strand Design for Biomolecular Computation", *Theoretical Computer Science*, vol. 287, pp.39-58, 2002.

[3] S.-Y. Shin, I.-H. Lee, D. Kim, and B.-T. Zhang, Multiobjective Evolutionary Optimization of DNA Sequences for Reliable DNA Computing", *IEEE Transactions on Evolutionary Computation*, vol. 9(20), pp.143-158, 2005.

[4] F. Tanaka, A. Kameda, M. Yamamoto, and A. Ohuchi, Design of Nucleic Acid Sequences for DNA Computing based on a Thermodynamic Approach, Nucleic Acids Research, 33(3), pp.903-911, 2005.

[5] J. Santalucia, " A Unified View of polymer, dumbbell, and oligonucleotide DNA nearest neighbor thermodynamics", *Proc. Natl. Acad. Sci., Biochemistry*, pp. 1460-1465, February 1998.

[6] A. D'yachkov, P.L. Erdös, A. Macula, V. Rykov, D. Torney, C-S. Tung, P. Vilenkin and S. White, "Exordium for DNA Codes," *Journal of Combinatorial Optimization*, vol. 7, no. 4, pp. 369-379, 2003.

[7] R. Deaton, M. Garzon, R.C. Murphy, J.A. Rose, D.R. Franceschetti, and S.E. Jr. Stevens, "Genetic search of reliable encodings for DNA-based computation," *Proceedings of the First Annual Conference on Genetic Programming*, pp. 9-15, July 1996.

[8] Bishop, M. , Macula, A. , Pogozelski, W. , and Rykov, V. , "DNA Codeword Library Design", *Proc. Foundations of Nanoscience – Self Assembled Architectures and Devices, (FNANO)*, April 2005.

[9] Tulpan, D.C. , Hoos, H. , Condon, A. ,"Stochastic Local Search Algorithms for DNA Word Design", *Eighth International Meeting on DNA Based Computers(DNA8)*, June 2002.

[10] S. Houghten, D. Ashlock and J. Lennarz, "Bounds on Optimal Edit Metric Codes", *Brock University Tech. Rer.t # CS-05-07*, July 2005.

[11] O. Milenkovic and N. Kashyap, "On the Design of Codes for DNA Computing," *Lecture Notes in Computer Science*, pp. 100-119, Springer Verlag, Berlin-Heidelberg, 2006.

[12] R. Brualdi, and V. Pless, "Greedy Codes," *Journal of Combinatorial Theory Series A*, vol. 64, pp. 10-30, 1993.

[13] http://www.xilinx.com/

[14] http://www.annapmicro.com/

[15] D. Burns, K. May, T. Renz, and V. Ross, "Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis," *MAPLD International Conference*, 2005.

[16] P.D. Michailidis and K.G. Margaritis, "New Processor Array Architectures for the Longest Common Subsequence Problem," *The Journal of Supercomputing*, vol. 32, pp. 51-69, 2005.

[17] Y.C. Lin and J.W. Yeh, "A Scalabl and Efficient Systolic Algorithm for the Longest Common subsequence Problem," *Journal of Information Science and Engineering*, vol. 18, pp. 519-532, 2002

Hardware Acceleration for Thermodynamically Constrained DNA Code Generation

Qinru Qiu*, Prakash Mukre*, Morgan Bishop**, Daniel Burns**, Qing Wu*

*Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902

**.Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441

qqiu@binghamton.edu, pmukre1@binghamton.edu, Morgan.Bishop@rl.af.mil, Daniel.Burns@rl.af.mil, qwu@binghamton.edu

ABSTRACT. Reliable DNA computing requires a large pool of oligonucleotides that do not produce cross-hybridize. In this paper, we present a transformed algorithm to calculate the maximum weight of the 2-stem common subsequence of two DNA oligonucleotides. The result is the key part of the Gibbs free energy of the DNA crosshybridized duplexes based on the nearest-neighbor model. The transformed algorithm preserves the physical data locality and hence is suitable to be implemented using a systolic array. A novel hybrid architecture that consists of a general purpose microprocessor and a hardware accelerator for accelerating the discovery of DNA under thermodynamic constraints is designed, implemented and tested. Experimental results show that the hardware system provides more than 250X speed-up compared to a software only implementation.

1. INTRODUCTION

A single DNA strand (i.e. oligonucleotides) is a sequence of four possible nucleotides denoted as A, C, G and T. Short DNA sequences can be synthesized easily and be used for different applications, including high density information storage [2], molecular computation of hard combinatorial problems [1], and molecular barcode to identify individual modules in complex chemical libraries [3]. These applications rely on the specific hybridization between DNA code word and its Watson-Crick complement. The key to success in DNA computing is the availability of a large collection of DNA code word pairs that do not crosshybridize.

The capability of hybridization between two oligonucleotides is determined by the base sequences of the hybridizing oligonucleotides, the location of potential mismatches, the concentrations of the molar strand, the temperature of the reaction and the length of the sequences [4]. The *melting temperature* (T_m) is a parameter that characterizes these factors [4]. It is defined as the temperature at which 50% of the DNA molecules have been separated to single strand. Another closely related measure of the relative stability of a DNA duplex is its Gibbs free energy denoted as ΔG^O. The nearest-neighbor (NN) model [8][12] was proven to be effective and accurate estimation for the free energy. In [14], the concept of *t-stem block insertion-deletion codes* was introduced that captures the key aspects of the nearest neighbor model. In the same reference, a dynamic programming algorithm is presented to calculate the maximum weight of the t-stem common subsequence.

Search methods for DNA codes are extremely time-consuming [5], and this has limited research on DNA codeword design, especially for codes of length greater than about 12-14 bases. For example, the largest known DNA codeword library generated based on the edit distance constraint with length 16 and edit distance 10 consists of 132 pairs. Composing such codes can take several days on a cluster of 10 G5 processors.

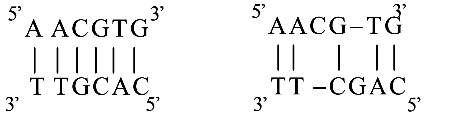
In [9], we presented a novel accelerator for the composition of reverse complement, edit distance DNA codes of length 16. It incorporates a hardware GA, hardware edit distance calculation, and hardware exhaustive search which extends an initial codeword library by doing a final scan across the entire universe of possible code words. The proposed architecture consists of a host PC, a hardware accelerator implemented in reconfigurable logic on a *field programmable gate array* (FPGA) and a software program running in a host PC that controls and communicates with the hardware accelerator. The proposed architecture using a modified genetic algorithm that uses a locally exhaustive, mutation-only heuristic tuned for speed. The proposed architecture successfully reduced the DNA codeword search time from 6 days (on 10 Pentium processors) to 1.5 hours and achieved an effective 1000X speed-up.

The edit distance metric only provides the first order approximation of the free energy of the DNA duplexes. To improve the quality of the code words, more accurate metric based on the thermodynamics of DNA duplexes must be considered. This paper focuses on implementing the nearest-neighbor based free energy calculation on the reconfigurable hardware accelerator. We present a transformed algorithm to calculate the maximum weight of the 2-stem common subsequence of two DNA oligonucleotides. The result is the key part of the Gibbs free energy of the DNA crosshybridized (CH) duplexes based on the nearest-neighbor model. The transformed algorithm preserves the physical data locality and hence is suitable to be implemented using systolic array. A new hardware accelerator for accelerating the discovery of DNA under thermodynamic constraints is further presented. The proposed architecture provides more than 250X speed-up compared to a software only implementation.

The remainder of this paper is organized as follows: Section 2 provides the necessary biological background and the nearest-neighbor model for free energy calculation. Section 3 introduces the weighted t-stem block insertion-deletion codes. Section 4 gives the transformed algorithm and its hardware implementation using 2D systolic array. Section 5 presents our problem formulation and the solution technique in hardware GA. Section 6 provides a performance comparison between the software version and the hardware version of the codeword search. Section 7 presents final conclusions.

2. BACKGROUND

The DNA molecule is a nucleic acid. It consists of two oriented oligonucleotide sequences. One end of it is denoted as 3' and the other as 5'. There are four types of bases, denoted briefly as A, T, C, and G. Each base can pair up with only one particular base through hydrogen bonds: A+T, T+A, C+G and G+C. Sometimes we say that A and T are complementary to each other while C and G are complementary to each other. A *Watson-Crick complement* of a DNA sequence is another DNA sequence which replaces all the A with T or vise versa and replaces all the T with A or vise versa, and also switches the 5' and 3' ends. A DNA sequence binds most stably with its Watson-Crick complement and the structure they form is called *Watson-Crick* (*WC*) *duplex*. Figure 1 (a) shows an example of a WC duplex. We refer to the non-WC duplex as *crosshybridized* (*CH*) *duplex*. Figure 1 (b) shows an example of a CH duplex. Only WC duplexes are needed during DNA computing. Therefore, it is important to design the DNA codes such that a fixed temperature can be found that is well above the melting point of all CH duplexes and well below the melting point of all WC duplexes that can form from strands in the code.



(a) WC duplex (b) CH duplex

Figure 1 WC duplex and CH duplex

The thermodynamics of binding of nucleic acids has been widely studied and reported in the literature. The nearest-neighbor (NN) model [12] was proven to be effective and accurate for the thermodynamic energy estimation. The NN model assumes that stability of a DNA duplex depends on the identity and orientation of neighboring base pairs. There are 10 possible NN pairs: AA/TT, AT/TA, TA/AT, CA/GT, GT/CA, CT/GA, GA/CT, CG/GC, GC/CG, and GG/CC. Based on the NN model, the total free energy change of a DNA duplex at temperature *T* can be calculated by the following equation:

$$\Delta G^{o}{}_{T}(total) = \Delta G^{o}{}_{T,initiation} + \Delta G^{o}{}_{T,symmetry} + \sum_{i \in Watson-Crick \text{ NNs}} G^{o}{}_{T,stack}(i) + \Delta G^{o}{}_{T,AT} \text{ Terminal},$$

where $\Delta G^{o}{}_{T,initiation}$ is the initiation energy, $\Delta G^{o}{}_{T,symmetry}$ is a parameter that reflects whether the duplex is self-complementary, $\Delta G^{o}{}_{T,AT\,\text{Terminal}}$ is a parameter that accounts for the differences between duplexes with terminal AT versus terminal GC, $\Delta G^{o}{}_{T,stack}(i)$ gives the thermodynamic energy of Watson-Crick NN duplex *i*. Their values at 37^{o}C are given in Table 1.

Example: Using Table 1, the NN free energy of DNA duplex $\begin{array}{c} 5'\text{–}CGTTGA-3' \\ 3'\text{–}GCAACT-5' \end{array}$ can be calculated as:

$$\textstyle\sum_{i\in \text{WC NNs}} \Delta G^{o}{}_{T,stack}(i) = \Delta G_{37,stack}(CG) + \Delta G_{37,stack}(GT) + \Delta G_{37,stack}(TT) + \Delta G_{37,stack}(TG) + \Delta G_{37,stack}(GA)$$

$$= -2.17 - 1.44 - 1.00 - 1.45 - 1.3 = -5.35 \text{ kcal mol}^{-1}.$$

Table 1 Nearest-neighbor thermodynamic parameters for DNA Watson-Crick pairs at 37^{O}C [12]

	A	C	G	T
A	-1	-1.44	-1.28	-0.88
C	-1.45	-1.84	-2.17	-1.28
G	-1.3	-2.24	-1.84	-1.44
T	-0.58	-1.3	-1.45	-1.00

While the parameters $\Delta G^{o}{}_{T,initiation}$, $\Delta G^{o}{}_{T,symmetry}$, and $\Delta G^{o}{}_{T,AT\,\text{Terminal}}$ can be obtained in a straightforward manner, the NN free energy (i.e. $\sum_{i\in WC\,\text{NN}} \Delta G^{o}{}_{T,stack}(i)$) is determined by the structure of the primary sequence of the DNA duplex. This work focuses on accelerating the calculation of NN free energy using reconfigurable hardware and applies it to hardware based DNA code word search.

3. T-STEM BLOCK INSERTION-DELETION CODES

In the rest of the paper, we adopt some notations that are used in reference [14]. We use [*n*] to denote the set {0, …, *n*-1} and (*n*) to denote the sequence 1, 2, …, *n*. We call $\alpha \prec (n)$ a string if and only if it is a subsequence of consecutive integers, e.g., $\alpha = i, i+1, \ldots, i+k$. Let $[q]^{n}$ denote the set of sequences of length *n* with entries in [*q*]. For $x = x_1, \ldots, x_n$ with $x \in [q]^{n}$ and $\sigma = i_1, i_2, \ldots, i_k$ where $\sigma \prec (n)$, we use $x_{\sigma} \prec x$ to denote the subsequence $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ and *x*[*i*] to denote the *i*th entry in sequence *x*. Given a non-negative real-valued function, Ω, on [*q*], we define the weight of subsequence x_{σ} as $\|x_{\sigma}\|_{\Omega} = \sum_{i\in\sigma} \Omega(x_i)$.

For $\sigma \prec (n)$, a substring $\beta \prec \sigma$ is called a block of σ if β is not a subsequence of any substring α of σ with $\beta \neq \alpha$. Denote σ as a sequence of blocks $\beta_1, \beta_2, \ldots, \beta_i, \ldots, \beta_l$, if the difference between the endpoint of β_i and the starting point of β_{i+1} is greater than or equal to *t*, then σ is a *t-gap sequence* of (*n*). It is denoted as $\sigma \in G_t(n)$. Given $\sigma \prec (n)$, $\tau \prec (m)$ with $|\sigma| = |\tau|$ and $\sigma \in G_t(n), \tau \in G_t(m)$, let $f : \sigma \to \tau$ be a unique mapping, σ and τ are said to be *t-gap block isomorphic* (denoted as $\sigma \underset{t}{\cong} \tau$) if $\alpha \prec \sigma$ is a string $\Leftrightarrow$ $f(\alpha) \prec \tau$ is a string. For $x \in [q]^{n}$ and

$y \in [q]^m$, if $x_\sigma = y_\tau$ and $\sigma \underset{t}{\cong} \tau$, then we say x_σ and y_τ are t-gap block isomorphic and denote it as $x_\sigma \underset{t}{\cong} y_\tau$.

Definition 1 For $2 \le t \le n-1$ and $x, y \in [q]^n$, we define the *weight of the longest common t-gap block subsequence* of *x* and *y* as $\phi^t_{\Omega,q}(x,y) \equiv \max\{\|x_\sigma\|_\Omega : x_\sigma \underset{t}{\cong} y_\tau\}$. The *weighted distance* of two t-gap block insertion-deletion codes is defined as $\Phi^t_{\Omega,q}(x,y) \equiv \min(\|x\|_\Omega, \|y\|_\Omega) - \phi^t_{\Omega,q}(x,y)$. When $t = 1$ and $\|x_\sigma\|_\Omega = |x_\sigma|$, $\Phi^1_{\Omega,q}(x,y)$ is the Levenshtein insertion-deletion distance.

The weight of the longest common t-gap block subsequence of *x* and *y* (i.e. $\phi^t_{\Omega,q}(x,y)$) can be calculated using dynamic programming. For $x, y \in [q]^n$ and $t \le i, j \le n$, let $M^t_{\Omega,i,j}$ denote $\phi^t_\Omega(x_{[1,i]}, y_{[1,j]})$ and $suf(x,y)$ denote the length of the longest common suffix between *x* and *y*. It is proved ([14]) that:

$$M^t_{\Omega,i,j} = \max(M^t_{\Omega,i-1,j}, M^t_{\Omega,i,j-1}, D^t_{\Omega,i,j}\},$$

(1)

where $D^t_{\Omega,i,j}$ is defined as

$$D^t_{\Omega,i,j} = \begin{cases} \max\left\{\|x_{[i-r+1,i]}\|_\Omega + M^t_{\Omega,i-r-t+1,j-r-t+1} : 1 \le r \le suf(x_{[1,i]}, y_{[1,j]})\right\} & if\ suf(x_{[1,i]}, y_{[1,j]}) \ge 1 \\ 0 & \text{otherwise} \end{cases}$$

. (2)

Given two sequences $x, y \in [q]^n$, $x_\sigma = y_\tau$ with a unique mapping $f : \sigma \to \tau$, a *t-stem* exists if and only if subsequences $x_{[i,i+t-1]} = y_{[f(i), f(i)+t-1]}$. Let σ^t_τ be the sequence of the first index of all the t-stems. For $x \in [q]^n$, let $d_q(x_{[a,a+t-1]})$ be a unique number in $[q^t]$ to represent $x_a x_{a+1} \ldots x_{a+t-1}$, we define $x^{(t)} \in [q^t]^{n-t}$ as a sequence whose *i*th element is equal to $(d_q(x_{i,i+t-1}))$. For $2 \le t \le n-1$, it can be proved that if $|\sigma^t_\tau| \ne 0$, then $x^{(t)}_{\sigma^t_\tau} \underset{t}{\cong} y^{(t)}_{f(\sigma^t_\tau)}$.

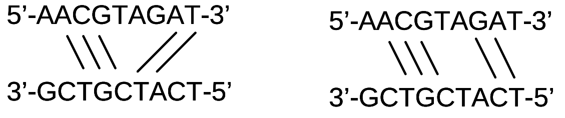
Definition 2 Let Ω be a weight function on $[q^t]$, the *maximum weighted t-stem common subsequence* is defined as $\psi^t_\Omega(x,y) = \max\left\{\left\|x^{(t)}_{\sigma^t_\tau}\right\|\right\}$. The *t-stem code distance* is defined as $\Psi^t(x,y) = \min\left(\left\|x^{(t)}\right\|_\Omega, \left\|y^{(t)}\right\|_\Omega\right) - \psi^t_\Omega(x,y)$.

It is proved ([14]) that the maximum weighted t-stem common subsequence of *x* and *y* is equal to the weight of the longest common t-gap block subsequence of $x^{(t)}$ and $y^{(t)}$, i.e. $\psi^t_\Omega(x,y) = \phi^t_{\Omega,q^t}\left(x^{(t)}, y^{(t)}\right)$.

Let the CH duplex between *x* and $\bar{y}$ be denoted as $x : \overleftarrow{y}$, where $\bar{y}$ is the WC complement of *y* and $\overleftarrow{y}$ is a representation of $\bar{y}$ in reversed order. If the duplex $x : \overleftarrow{y}$ have a secondary structure, then its free energy of nearest neighbor stacked pairs can be calculated as $\psi_{\Omega}^{2}(x,y)$, where the weight function Ω is equal to $-\Delta G_{T,stack}^{o}$.

Example: Consider the CH duplex $\begin{matrix} 5'\text{–}AACGTAGAT-3' \\ 3'\text{–}GCTGCTACT-5' \end{matrix}$. It corresponds to strings *x* = *AACGTAGAT* and *y* = *CGACGATGA*. Because $x_{[2,4][8,9]} = y_{[3,5][6,7]}$, we have $\sigma = [2,4][8,9]$, $\tau = [3,5][6,7]$, $\sigma_{\tau}^{2} = 2,3,8$, and $\left\| x_{\sigma}^{(2)} \right\| = \left\| x_{2,3,8}^{(2)} \right\| = -\Delta G_{37,stack}(AC) - \Delta G_{37,stack}(CG) - \Delta G_{37,stack}(AT) = 4.49\, kcal/mol$.

Let *A*, *C*, *G*, *T* be encoded as 0, 1, 2, and 3, then *x* = 0, 0, 1, 2, 3, 0, 2, 0, 3 and *y* = 1, 2, 0, 1, 2, 0, 3, 2, 0. $x^{(2)}$ = 0, 1, 6, 11, 12, 2, 8, 3 and $y^{(2)}$ = 6, 8, 1, 6, 8, 3, 14, 8. It is easy to see that $x_{2,3,8}^{(2)} = y_{3,4,6}^{(2)}$. Let $\sigma = 2,3,8$ and $\tau = 3,4,6$, because any string in σ corresponds to a string in τ, and the gaps between blocks in σ and τ are equal to 2, we say $\sigma \underset{2}{\cong} \tau$ and $x_{\sigma}^{(2)} \underset{2}{\cong} y_{\tau}^{(2)}$. Note that although $x_{2,3,7,8}^{(2)} = y_{3,4,5,6}^{(2)}$, because a string in $\tau = 3,4,5,6$ does not necessarily correspond to a string in $\sigma = 2,3,7,8$, therefore, $x_{2,3,7,8}^{(2)}$ and $y_{3,4,5,6}^{(2)}$ are not t-gap block isomorphic. Using equation (1) and (2) we can find that $\psi_{\Omega}^{2}(x,y) = \phi_{\Omega}^{t}(x^{(2)}, y^{(2)}) = \left\| x_{2,3,7}^{(2)} \right\| = -\Delta G_{37,stack}(AC) - \Delta G_{37,stack}(CG) - \Delta G_{37,stack}(GA) \;\; = 4.91\, kcal/mol$. Figure 2 shows the secondary structures in the CH duplex that for $\sigma_{\tau}^{2} = 2,3,8$ and $\sigma_{\tau}^{2} = 2,3,7$.



(a) Secondary structure corresponding to $\sigma_{\tau}^{2} = 2,3,8$

(b) Secondary structure corresponding to $\sigma_{\tau}^{2} = 2,3,7$

Figure 2 Represent DNA secondary structure using 2-stem insertion-deletion codes

In this work, we estimate the NN free energy of a CH duplex by calculating their maximum weighted 2-stem common subsequence. In the next section, we will present a dynamic programming algorithm that is suitable for a 2D systolic array implementation.

4. CALCULATION OF NN FREE ENERGY USING 2D SYSTOLIC ARRAY

Based on the equations (1) and (2), a dynamic programming algorithm was developed.

Given a CH duplex $x : \overset{\leftarrow}{y}$, we define 3 matrices. They include a *suffix matrix* (**S**) which stores the length of the longest common suffix between *x* and *y*, a *weighted suffix matrix* (**WS**) which stores the accumulated weight of each common stem-2 and an energy matrix (**E**) which stores the accumulated free energy of the possible NNs. The value of the *ij*th entry of these matrices can be calculated using the following equations.

$$s_{ij} = \begin{cases} s_{i-1,j-1} + 1 & \text{if } x[i] = y[j] \\ 0 & \text{otherwise} \end{cases},$$

(3)

$$ws_{i,j} = \begin{cases} ws_{i-1,j-1} + w(x[i-1], x[i]) & \text{if } x[i] = y[j]\ \&\ x[i-1] = y[i-1] \\ 0 & \text{otherwise} \end{cases},$$

(4)

$$e_{ij} = \begin{cases} \max(ws_{i,j} - ws_{i-1,j-1} + e_{i-2,j-2}, ws_{i,j} - ws_{i-2,j-2} + e_{i-3,j-3}, & \\ \quad ....., ws_{i,j} - ws_{i-s_{ij},j-s_{ij}} + e_{i-s_{ij}-1,j-s_{ij}-1},\ e_{i,j-1}, e_{i-1,j}) & \text{if } x[i] = y[j]\ . \\ \max(e_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) & \text{otherwise} \end{cases} \qquad (5)$$

The parameter $w(a[i\text{-}1],a[i])$ is the stack-pair free energy specified in Table 1. The bottom right entry of the ***E*** matrix gives the NN free energy of $x : \overleftarrow{y}$.

Example: Consider x = 5'*AATGA*3' and $\overleftarrow{y} = 3'CATGG5'$ (i.e. $y = 5'GTACC3'$,) the matrix **S**, **WS**, and **E** can be calculated as the following and the NN free energy of the CH duplex is 2.33.

$$\mathbf{S} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \mathbf{WS} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.88 & 0 & 0 \\ 0 & 0 & 0 & 2.33 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{E} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.88 & 0.88 & 0.88 \\ 0 & 0 & 0.88 & 2.33 & 2.33 \\ 0 & 0 & 0.88 & 2.33 & 2.33 \end{bmatrix}.$$

Systolic array processing has been widely used in parallel computing to enhance performance. Its general architecture is given in Figure 3 (b). It has *N*×*N* connected processors. Each processor performs an elementary calculation. The processor *P*(*i*,*j*) reads data from its up stream neighbors *P*(*i*-1,*j*), *P*(*i*,*j*-1) and *P*(*i*-1, *j*-1), and propagates the results to its down stream neighbors *P*(*i*+1,*j*), *P*(*i*,*j*+1) and *P*(*i*+1, *j*+1). After an initialization period that is needed to fill the pipeline, a systolic array generates one result per 2 clock periods.

Equation (3)~(5) cannot be directly mapped to a 2D systolic array architecture because to calculate e_{ij} we need the value of $ws_{i-d,j-d}$ ($e_{i-d,j-d}$), $1 \le d \le s_{ij}$. The variable e_{ij} is calculated by processor *P*(*i*,*j*). The variables $ws_{i-d,j-d}$ and $e_{i-d,j-d}$ are calculated by processor *P*(*i-d*, *j-d*). If the calculation of e_{ij} is performed at clock period *t*, then the calculations of $ws_{i-d,j-d}$ and $e_{i-d,j-d}$ of the same DNA duplex are performed at clock period $t-2d$. Because cells in the systolic array will register the new input and update their results every 2 clock periods, it is not possible for us to access the data of $ws_{i-d,j-d}$ and $e_{i-d,j-d}$ at clock period *t* if *d* is greater than 1. One way to handle this problem is to memorize the values of $ws_{i-d,j-d}$ and $e_{i-d,j-d}$ by adding extra storage elements. Because the maximum value of s_{ij} can be as high as the length of the DNA strand, which in our case is 16, this solution requires us to duplicate each cell in the systolic array 16 times. This is not practical as it significantly increases the hardware cost.

In this work, we use function transformation to simplify the hardware design. We define a *minimum weighted suffix matrix* (**MIN_WS**) which stores the minimum value of the difference between $ws_{i-d,j-d}$ and $e_{i-d-1,j-d-1}$, where $1 \le d \le s_{ij}$. The *ij*th entry of **MIN_WS** can be calculated as

$$min\_ws_{ij} = \begin{cases} \min(min\_ws_{i\text{-}1,j\text{-}1}, ws_{ij} - e_{i-1,j-1}) & \text{if } x[i] = y[j] \\ 1{,}000{,}000 & \text{otherwise} \end{cases}, \qquad (6)$$

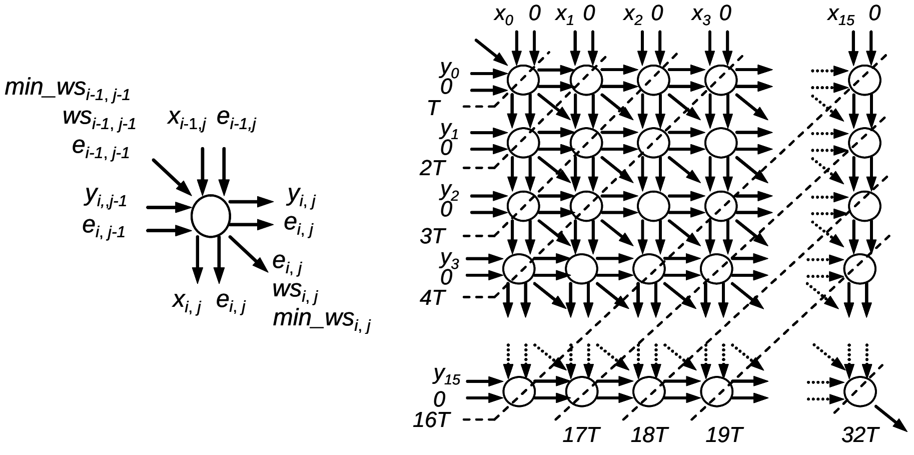
when $x[i] \ne y[j]$, min_ws_{ij} will be set to an extremely large number, otherwise, it is the minimum between $min\_ws_{i\text{-}1,j\text{-}1}$ and ws_{ij}-$e_{i\text{-}1,j\text{-}1}$. The calculation of e_{ij} and ws_{ij} is transformed into the following equations.

$$ws_{i,j} = \begin{cases} ws_{i-1,j-1} + w(x[i-1], x[i]) & \text{if } x[i] = y[j]\ \&\ min\_ws_{ij} \ne 1{,}000{,}000 \\ 0 & \text{otherwise} \end{cases} \qquad (7)$$

$$e_{ij} = \begin{cases} \max(ws_{i,j} - min\_ws_{i\text{-}1,j\text{-}1}, e_{i,j-1}, e_{i-1,j}) & \text{if } x[i] = y[j] \\ \max(e_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) & \text{otherwise} \end{cases}. \quad (8)$$

Equations (6)~(8) are equivalent to equations (3)~(5), however, only information from adjacent cells is needed in the calculation, hence, they can be implemented using the systolic array architecture.

The hardware design of the 2D systolic array can be derived directly from equations (6)~(8). The systolic array is an $n\times n$ array of identical cells. Each cell in the array has 7 inputs, among which the inputs $e_{i\text{-}1,j}$ and x[i-1, j] are coming from the cell that is located above, the inputs $e_{i,j\text{-}1}$ and y[i, j-1] are coming from the cell that is located to the left, and the inputs $e_{i\text{-}1,j\text{-}1}$, $ws_{i\text{-}1,j\text{-}1}$ and $min\_ws_{i\text{-}1,j\text{-}1}$ are coming from the cell that is located to the upper left. Each cell performs the computations that are described in equations (6)~(8). For cell (i,j), the outputs $x_{i,j}$ and $y_{i,j}$ are equal to the inputs $x_{i\text{-}1,j}$ and $y_{i,j\text{-}1}$. Figure 3 (a) gives the structure of each cell, including its input/output and the computation implemented. The variable $x_{i,j}$ and $y_{i,j}$ are represented as 2 bit binary numbers with A=00, C=01, G=10, and T=11. The variable $e_{i,j}$, $ws_{i,j}$ and $min\_ws_{i,j}$ are represented as 14 bit signed integer numbers.



(a) Cell architecture (b) 2D systolic array

Figure 3 Calculating maximum weighted 2-stem common subsequence using 2D systolic array

The overall architecture of the 2D systolic array as well as the data dependency and timing information are shown in Figure 3 (b). In order to prevent ripple through operation, the cells in the even columns and even rows or odd columns and odd rows are synchronous to each other and perform the computation in the same clock period. The rest of the cells are also synchronous to each other but perform the computation in the next clock period. In this way, the results propagate through the array diagonally.

5. PROBLEM FORMULATION AND SOLUTION TECHNIQUE

We consider each DNA codeword as a sequence of length n in which each symbol is an element of an alphabet of 4 elements. Let $G(x:\overleftarrow{y})$ denote the nearest neighbor free energy of duplex $x:\overleftarrow{y}$. In this work, we focus on searching for a set of DNA codeword pairs S, where S consists of a set of DNA strands of length n and their reverse complement strands e.g. {($s_1, \overline{s_1}$), (s_2, $\overline{s_2}$), …}, where (s_1, $\overline{s_1}$) denotes a strand and its Watson-Crick complement. The problem can be formulated as the following constrained optimization problem:

$$\max|S| \quad \text{such that} \quad (8)$$

$$g - range \le \max\left(G(s_1:\overleftarrow{s_1}), G(\overline{s_1}:\overleftarrow{\overline{s_1}})\right) \le g, \quad (9)$$

$$g - range \le \max_{s_2 \in S, s_2 \ne s_1} \left(G(s_1 : \overleftarrow{s_2}), G(s_1 : \overleftarrow{\overline{s_2}}), G(\overline{s_1} : \overleftarrow{s_2}), G(\overline{s_1} : \overleftarrow{\overline{s_2}}) \right) \le \mathrm{g} \qquad (10)$$

where *g* and *range* are user defined threshold called *CH upper bound* and *CH range*. Equation (8) indicates that our objective is to maximize the size of the DNA codeword library. Constraints (9)~(10) specify that the NN free energy of any CH duplexes must be lower than or equal to *g* but greater than or equal to *g-range*. For any DNA duplex, the weakest stacked pair is the AT pair with $\Delta G^{o}_{37,stack}(AT) = 0.88$ and $\Delta G^{o}_{37,stack}(TA) = 0.58$. Therefore, for $y, x \prec [A, C, G, T]^{16}$, $\min(\left\| x^{(2)} \right\|_{\Omega}, \left\| y^{(2)} \right\|_{\Omega}) = 7*(0.58+0.88)+0.58 = 10.8$. Based on definition 2, the weighted t-stem distance between x and y is greater than $10.8 - g$ and less than $10.8 - g + range$ if $g - range \le \psi^{2}_{\Omega}(x, y) = G(x : \overleftarrow{\overline{y}}) \le g$. Therefore, constraints (9)~(10) ensure that the t-stem distance between any non-WC pairs in the library is within the range [10.8-*g*+*range*, 10.8-*g*]. The range was initially introduced because we thought that adding code words that are too far away from the rest of the library will restrict the future growth of the library. Therefore, we only add code words that are "just good enough". Later in the experiments we found that the range has little impact on the size of the library, however, it has a significant impact on the convergence speed of the GA.

The optimization problem is solved using a genetic algorithm. A genetic algorithm (GA) is a stochastic search technique based on the mechanism of natural selection and recombination. Solutions, which are also called *individuals*, are evolved from generation to generation, with *selection*, *mating*, and *mutation* operators that provide an effective combination of exploration of the global search space.

Given a codeword library *S*, the fitness of each individual *d* reflects how well the corresponding codeword fits into the current codeword library. Two values define the fitness, *reject_num* and *max_match*. The *reject_num* is the number of codewords in the library which does not satisfy the condition (9)~(10) and $\max\_match = \max_{s_2 \in S, s_2 \ne s_1} \left(G(s_1 : \overleftarrow{s_2}), G(s_1 : \overleftarrow{\overline{s_2}}), G(\overline{s_1} : \overleftarrow{s_2}), G(\overline{s_1} : \overleftarrow{\overline{s_2}}) \right)$.

A traditional GA mutation function might randomly pick an individual in the population, randomly pick a pair of bits in the individual representing one of its 16 bases, and randomly change the base to one of the 3 other bases in the set of 4 possible bases. In the proposed algorithm, however, we randomly select an individual, but then to exhaustively check all of the 48 possible base changes. This is an attempt to speed beneficial evolution of the population by minimizing the overhead that would be associated with randomly picking this individual again and again in order to test those mutations. We also specify that if none of the 48 mutations were beneficial, a random individual will be generated to replace the original one. For more details about the genetic algorithm and its hardware implementation, please refer to [9][11]. In this work, we extend the architecture of the hardware GA presented in [9] to incorporate the consideration of nearest-neighbor free energy. The 2D systolic array that is presented in section 4 is used as fitness evaluation module and the main state machine controller of the GA is modified so that it checks all the constraints (9)~(10).

6. EXPERIMENTAL RESULTS

A hardware accelerator that uses a stochastic GA to build DNA codeword libraries of codeword length 16 has been designed, implemented, and tested. The design was implemented on the reconfigurable computing platform that is composed of a desktop computer and an Annapolis WildStar–Pro FPGA board [10]. The FPGA board is plugged into the PCI-X slot of the host system. The WildStar-Pro uses XC2VP70 FPGA that has 74,448 programmable logic cells. The hardware accelerator uses about 80% of the logic resource, and it runs at 45 MHz clock frequency. A hardware based code extender that uses exhaustive search to complete the codeword library generated from GA has also been designed and implemented. All the code word libraries that have been found were verified using the online tool SynDCode[13]. Since GA is a stochastic algorithm, all results reported are the average of 5 runs.

The first set of experiments compares the performance of the hardware-based and the software-only DNA codeword search. Two search algorithms are implemented. They are denoted as “deterministic search” (DS) and “randomized search” (RS). The population size is 16. The population of the DS was initialized using 16 sequential data from 0x000003F0 to 0x000003FF, which corresponds to DNA codeword 3’AATTTAAAAAAAAAAA’5 and 3’TTTTTAAAAAAAAAAA’5, while the population of the RS was initialized randomly. When a new codeword is found, or when none of the mutated codewords has lower fitness than the original individual, a new individual will be generated to replace the original one. In the DS, a counter is used to generate the new individual. The counter is initialized to 0x000006D6. In the RS, the new individual is generated randomly. The random search is more effective than the deterministic search. However, in order to compare the speed of hardware-based implementation and software-based implementation, we must ensure that the two systems perform exactly the same computation tasks. This is achievable only with a deterministic algorithm. All experiments were run with *g* = 8.5 and *range* = 1.0. They are terminated after 300 code word pairs have been found.

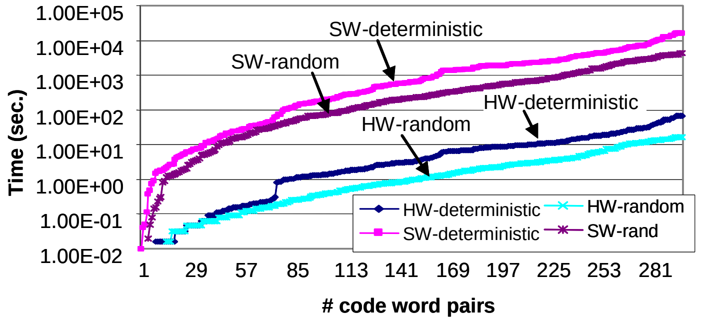
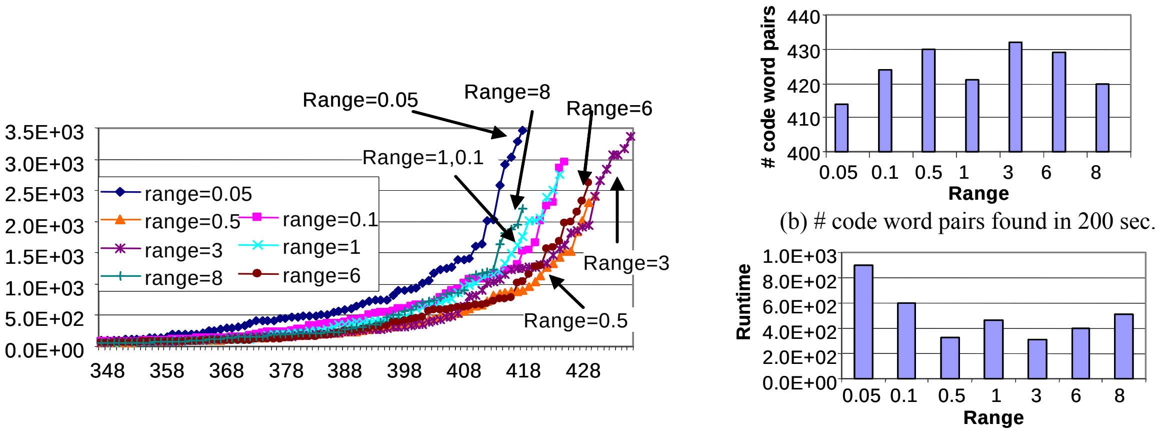


Figure 4 Comparison between hardware-based and software-based implementation

Figure 4 shows the average time it takes to build large thermodynamic constrained DNA code word libraries using software on a single processor workstation, and using the hardware accelerator. The lower curve indicates faster speed. As we can see, the software-based deterministic search has the lowest performance, while the hardware-based random search has the highest performance. The hardware-based deterministic search provides approximately 240X speed-up compared to the software-only version while the hardware-based random search provides approximately 260X speed-up compared to the software-only version. Compared to the deterministic search, random search provides approximately 3.7X and 4X speed-ups using software-only and hardware-based implementations, respectively. The plot also shows that the curves for software-only implementation and hardware-based implementation are almost parallel to each other, which indicates that they both have the same complexity. Therefore, the performance gain that has been achieved by using hardware acceleration is a constant ratio.

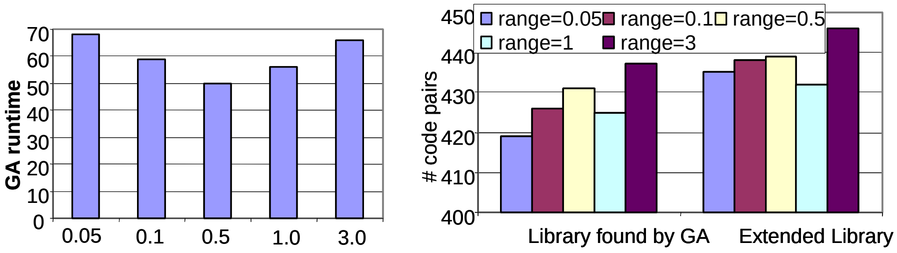


(b) # code word pairs found in 200 sec.

(a) Impact of *range* on the speed of the code word search

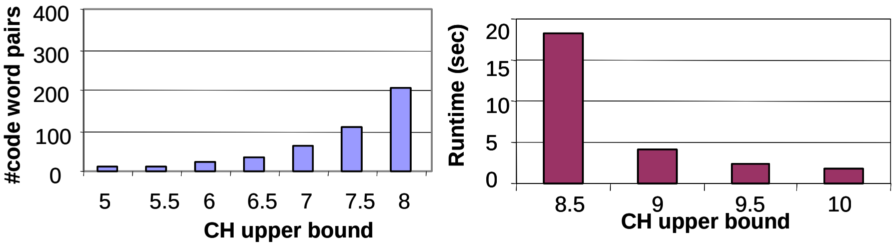
(c) Time to find 400 code word pairs

Figure 5 Impact of different ranges on the search speed



(a) Runtime of GA under different range (b) Library size under different range

Figure 6 Impact of different ranges on the library size



(a) # code word pairs found in 5 minutes (b) Time to find 300 code word pairs

Figure 7 Code word search under different CH upper bound

The second set of experiments evaluated the impact of CH range on the speed and quality (library size vs time) of the search. We varied the *CH range* from 0.05 to 3 and ran the GA based code word search with *g*=8.5. Figure 5 (a) shows library size vs time for different CH ranges. Figure 5 (b) and (c) give the number of code word pairs found in 200 seconds, and the time to find 400 code word pairs, for different CH ranges. The results show that the library growth is slower when the CH range is either too large or too small. In the next experiment, we ran the GA until it could not find any new code word for 10 minutes,) then we use exhaustive search to complete the codeword library. Figure 6 (a) shows the runtime of GA under different ranges and Figure 6 (b) shows the size of the library found by GA and the size of the final library. As we can see, the GA converges faster when setting the range to appropriate value. Compared to range = 0.5, the runtime of GA is 26% and 24% longer at range= 0.05 and 3.0 respectively. Opposite to our original believe, the distance range does not have significant impact on the library size. The sizes of libraries found by GA at different ranges only have 4% difference and the sizes of final libraries only have 3% difference. The exhaustive search usually finishes within 2 hours.

The third set of experiments compares the search speed for different CH upper bounds (*g*). We varied the CH upper bound from 6.5 to 10.0 and run the GA-based code word search. We stop the search when it found 300 code word pairs or the run time exceeds 15 minutes. Figure 7 (a) shows the number of code word pairs found in 5 minutes for CH upper bounds from 5 to 8.0 while Figure 7 (b) shows the runtime to find 300 code word pairs for CH upper bound from 8.5 to 10. The results indicate that as the CH upper bound increases, the chances to find a code word increases exponentially.

The significance of the hardware accelerator is that it enables us to evaluate different code word search algorithms and explore the lower bound of optimal code word libraries in a reasonable amount of time. For example, without the hardware accelerator, each experiment in the second set would take more than 20 days.

7. CONCLUSIONS AND FUTURE WORK

In this work, we propose a systolic array architecture to calculate the nearest-neighbor free energy of DNA duplexes. A hardware accelerator has been developed that searches for DNA codewords based on thermodynamic energy constraints. In the future, we plan to extend the current architecture to search and extend DNA code libraries with word lengths up to 32 bases.

8. REFERENCES

[1] L. M. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, vol. 266, pp. 1021-1024, November 1994.

[2] M. Mansuripur, P.K. Khulbe, S.M. Kuebler, J.W. Perry, M.S. Giridhar, and N. Peyghambarian, "Information Storage and Retrieval using Macromolecules as Storage Media," *Proceedings of Optical Data Storage*, 2003.

[3] S. Brenner and R. A. Lerner, "Encoded Combinatorial Chemistry," *Proc. Natl. Acad. Sci. USA*, vol 89, pp5381-5383, June 1992.

[4] R. Deaton and M. Garzon, "Thermodynamic Constraints on DNA-based Computing," *Computing with Bio-Molecules: Theory and Experiments*, Springer-Verlag.

[5] A. Brenneman and A. Condon, "Strand Design for Biomolecular Computation", *Theoretical Computer Science*, vol. 287, pp.39-58, 2002.

[6] S.-Y. Shin, I.-H. Lee, D. Kim, and B.-T. Zhang, "Multiobjective Evolutionary Optimization of DNA Sequences for Reliable DNA Computing", *IEEE Transactions on Evolutionary Computation*, vol. 9(20), pp.143-158, 2005.

[7] F. Tanaka, A. Kameda, M. Yamamoto, and A. Ohuchi, "Design of Nucleic Acid Sequences for DNA Computing based on a Thermodynamic Approach," *Nucleic Acids Research*, 33(3), pp.903-911, 2005.

[8] J. Santalucia, "A Unified View of polymer, dumbbell, and oligonucleotide DNA nearest neighbor thermodynamics", *Proc. Natl. Acad. Sci., Biochemistry*, pp. 1460-1465, February 1998.

[9] Qinru Qiu, D. Burns, Q. Wu and Prakash Mukre, "Hybrid Architecture for Accelerating DNA Codeword Library Searching*," to appear in Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, April 2007.

[10] http://www.annapmicro.com/

[11] D. Burns, K. May, T. Renz, and V. Ross, "Spiraling in on Speed-Ups of Genetic Algorithm Solvers for Coupled Non-Linear ODE System Parameterization and DNA Code Word Library Synthesis," *MAPLD International Conference*, 2005.

[12] J. SantaLucia, Jr. and D. Hicks, "The thermodynamics of DNA Structural Motifs," *Annu. Rev. Biophys. Biomol. Struct*. 33:415-40, 2004.

[13] M. A. Bishop, A. J. Macula1, T. E. Renz, "SynDCode: Cooperative DNA Code Generating Tool," *Proc. of 3rd Annual Conference of Foundations of Nanoscience*, April, 2006.

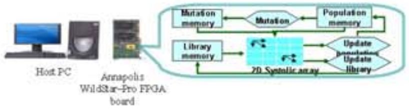
[14] A.G. D'yachkov, A.J. Macula, W.K. Pogozelski, T.E. Renz, V.V. Rykov, and D.C. Torney, "A Weighted Insertion-Deletion Stacked Pair Thermodynamic Metric for DNA Codes," *Lecture Notes in Computer Science*, Vol. 3384/2005, pp. 90-103, Springer Berlin/Heidelber

One the Hardware Acceleration of (Bioinformatics) Algorithms

Daniel J. Burns and Dr. Thomas E. Renz, Air Force Research Laboratory, Rome, NY burnsd@rl.af.mil , renzt@rl.af.mil ; Qinru Qiu, Qing Wu, Binghamton University, Binghamton, NY, qqiu@binghamton.edu , qwu@binghamton.edu ; Morgan Bishop, JEANSEE Corp., Geneseo, NY, bishopm@rl.af.mil ; Andrew C. Flack, University of Rochester, flacka@rl.af.mil

Abstract

- Fast Probe Set Design tools are needed to the support rapid development and update of multi-organism bio-threat detection microarrays, as well as individualized whole genome diagnostic microarray applications.
- Hardware accelerators have been developed by others for RC-BLAST heuristics based sequence alignment [1], the Smith-Waterman homology search algorithm [2], etc.
- Recently the authors have developed new hardware accelerators that are implemented in reconfigurable logic (FPGAs) for the Length of the Longest Common Substring (LLCS) [3] and the Gibbs free energy based on the nearest-neighbor model both for the case of 'short' DNA oligos (16mer x 16mer), and for the genetic algorithm (GA) optimization algorithm [4,5].
- Up to 1000x speedups have been achieved for the DNA Code generation problem (e.g. for composing synthetic tag-antitag (TAT) libraries, and sticky-edge DNA tiling schemes for self-assembly of nanostructures).
- Hardware acceleration of the Probe Set Design Problem involving high speed GA optimizations involving simultaneous LLCS and Gibbs energy calculations appear to be feasible for checking n=25 to n=50 mers against large filtered genomes with 1/n^2 speedups on modest PC platforms.



Calculating Weighted t-stem Distance

$$s_{ij}=\begin{cases} s_{i-1,j-1}+1 & \text{if } x[i]=y[j] \\ 0 & \text{otherwise}\end{cases}$$

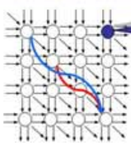
The longest suffix between $x_{[0,i]}$ and $y_{[0,j]}$

$$ws_{i,j}=\begin{cases} ws_{i-1,j-1}+w(x[i-1],x[i]) \\ \quad \text{if } x[i]=y[j]\ \&\ x[i-1]=y[i-1] \\ 0 \quad \text{otherwise}\end{cases}$$

Accumulated weight of each common stem-2

NN free energy between $x_{[0,i]}$ and $y_{[0,j]}$

$$e_{ij}=\begin{cases} \max(ws_{i,j}-ws_{i-1,j-1}+e_{i-2,j-2}, ws_{i,j}-ws_{i-2,j-2}+e_{i-3,j-3}, \\ \ldots\ldots, ws_{i,j}-ws_{i-s_{ij},j-s_{ij}}+e_{i-s_{ij}-1,j-s_{ij}-1}, e_{i,j-1}, e_{i-1,j}) \\ \qquad \text{if } x[i]=y[j] \\ \max(e_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) \qquad \text{otherwise}\end{cases}$$



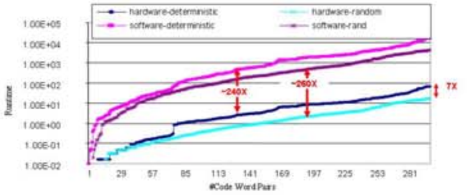
Calculates s_{ij}, ws_{ij} and e_{ij}

Each cell in a 2D systolic array only connects to its nearest neighbor

The algorithm requires remote data to calculate ws_{ij}, which increases the hardware complexity.

Experimental Results I

The hardware accelerator provides about 260X speed-ups compared to the software only implementation



Nearest Neighbor Model

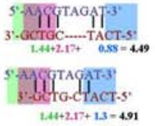
- The stability of a DNA duplex depends on the neighboring base pairs

$$\Delta G^{o}_{T}(total)=\Delta G^{o}_{T,initiation}+\Delta G^{o}_{T,symmetry}+\sum_{i\in Watson-Crick\ NNs} G^{o}_{T,stack}(i)+\Delta G^{o}_{T,AT\ Terminal}$$

Searching for an alignment that maximizes the free energy is time consuming, which requires hardware acceleration.

Nearest Neighbor Free Energy for WC Pairs

	A	C	G	T
A	-1	-1.44	-1.28	-0.88
C	-1.45	-1.84	-2.17	-1.28
G	-1.3	-2.24	-1.84	-1.44
T	-0.58	-1.3	-1.45	-1.00



Systolic Algorithm

$$\max_{1\le d\le s_{ij}}(ws_{i,j}-ws_{i-d,j-d}+e_{i-d-1,j-d-1})$$

$$\Updownarrow$$

$$ws_{i,j}-\underbrace{\min_{1\le d\le s_{ij}}(ws_{i-d,j-d}-e_{i-d-1,j-d-1})}_{min\_ws_{i-1,j-1}}$$

$$min\_ws_{ij}=\begin{cases} \min(min\_ws_{i-1,j-1}, ws_{ij}-e_{i-1,j-1}) & \text{if } x[i]=y[j] \\ 1{,}000{,}000 & \text{otherwise}\end{cases}$$

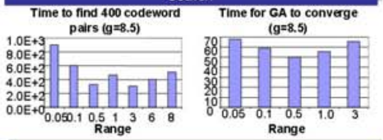
$$ws_{i,j}=\begin{cases} ws_{i-1,j-1}+w(x[i-1],x[i]) \\ \quad \text{if } x[i]=y[j]\ \&\ min\_ws_{ij}\neq 1{,}000{,}000 \\ 0 \quad \text{otherwise}\end{cases}$$

$$e_{ij}=\begin{cases} \max(ws_{i,j}-min\_ws_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) & \text{if } x[i]=y[j] \\ \max(e_{i-1,j-1}, e_{i,j-1}, e_{i-1,j}) & \text{otherwise}\end{cases}$$

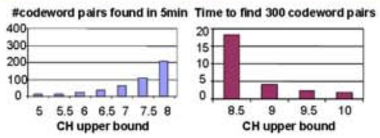
The transformed algorithm preserves the physical data locality and hence is suitable to be implemented using systolic array.

Experimental Results II

The value of CH range impacts the speed of codeword search



The search speed increases as the CH upper bound increases



T-Stem Block Insertion-deletion Codes

- Two sequences *x* and *y* have a *t*-stem if and only if there is a unique mapping $f:\sigma\rightarrow\tau$ such that $x_{[i,i+t-1]}=y_{[f(i),f(i)+t-1]}$
  - For nearest neighbor calculation, *t*=2
- The maximum weighted t-stem common subsequence between x and y is the maximum summation of the weight of all possible t-stems between x and y.
  - Dynamic programming and 2D HW systolic arrays available
- The NN stacked pair energy of DNA duplex *x:y'* is equal to the maximum weighted 2-stem between *x* and *y* where *y* is the WC complement of *y'*

Probe Set Design Problem

- Formulate Probe Set Design problem as a constrained optimization problem for intended and unintended probe-target bindings
  - *G(x:y)*: the NN free energy of *x* and *y*

Maximize |S| such that

$$g-range\le\max_{s_1\in S,s_2\neq s_1}\left(G(s_1:\overleftarrow{s_2}),G(s_1:\overleftarrow{\overline{s_2}}),G(\overline{s_1}:\overleftarrow{s_2}),G(\overline{s_1}:\overleftarrow{\overline{s_2}})\right)\le g$$

$$g-range\le\max\left(G(s_1:\overleftarrow{s_1}),G(\overline{s_1}:\overleftarrow{\overline{s_1}})\right)\le g,$$

- Genetic algorithm (GA) + exhaustive search (ES)
  - Hardware GA:
    - Random initialization of a small population of sets
    - Rank based selection, single point or uniform crossover mating, random or targeted mutation strategies, de-cloning, multi-deme HW GA available
  - Hardware Exhaustive Search Fitness Function
    - Scan the entire filtered target space
- 500x speedup would reduce a 10 day calculation to 30 minutes.

Conclusions

- A 2D systolic architecture is presented that calculates the maximum weight of 2-stem common subsequences.The design can be used to calculate the NN stacked pair free energy for DNA duplexes as well as the number of 2-stem common subsequences
- FPGA based hardware has been developed to accelerate Gibbs energy constrained DNA code searching using GA and exhaustive search
- 250X speed-up demonstrated over software only implementation
- 32x32 mer or 50x50 mer or larger arrays requires new design effort
- On-board memory access time not expected to bottleneck calculation time

References

[1] K. Muriki, K. Underwood, and R. Sass, "Rc-blast: Towards an open source hardware implementation," *Proceedings of 4th IEEE International Workshop on High Performance Computational Biology*, 2005.

[2] R. K. Karanam, A. Aavindran, A/ Mukherjee, C. Gibas, "Using FPGA-Based Hybrid Computers for Bioinformatics Applications", (Xilinx) Xcell Journal, Third Quarter, 2006.

[3] Q.Qiu, D. Burns, Q. Wu and P. Mukre, "Hybrid Architecture for Accelerating DNA Codeword Library Searching", *Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, April 2007.

[4] A.G. D'yachkov, A.J. Macula, W.K. Pogozelski, T.E. Renz, V.V. Rykov, and D.C. Torney, "A Weighted Insertion-Deletion Stacked Pair Thermodynamic Metric for DNA Codes," *Lecture Notes in Computer Science*, Vol. 3384/2005, pp. 90-103, Springer Berlin/Heidelber.

[5] Q.Qiu, P. Mukre, M. Bishop, D. Burns, Q. Wu, "Hardware Acceleration for Thermodynamically Constrained DNA Code Generation", *Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, April 2007.